お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

User's Manual

IMA アセンブラ・プログラミング・ガイド

（740 ファミリ） 第 2 版

ルネサス エレクトロニクス
www.renesas.com

Rev.2.00

COPYRIGHT NOTICE

© Copyright 1997 IAR Systems. All rights reserved.
No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.
In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

C-SPY is a trademark of IAR Systems. Windows and MS-DOS are trademarks of Microsoft Corp.
All other product names are trademarks or registered trademarks of their respective owners.
Fifth edition: June 1997
Part no: A740–2

本書は、株式会社ソフトボートが IAR Systems AB の許可を得て作成した A740 の日本語マニュアルです。本書の一部または全部を無断で複製、複写、転記することは、法律で禁じられていますので御注意ください。

株式会社ソフトボートは、IAR Systems AB の日本における総代理店です。

ウエルカム

740 アセンブラ・プログラミング・ガイドへようこそ。

本ガイド書は、740 マイクロプロセッサ用のアセンブラ、XLINK リンカおよび XLIB ライブラリアンに関するリファレンス情報を提供します。そしてこの情報は、組み込みワークベンチ版とコマンドライン版の両方のツールに適用されます。

組み込みワークベンチ用

組み込みワークベンチを使用する場合、「組み込みワークベンチ・インターフェース・ガイド」を読んで、組み込みワークベンチのコマンドとダイアログ・ボックスについて習熟するようにしてください。

コマンドライン用

コマンドライン版を使用する場合には、「コマンド・ライン・インターフェース・ガイド」を参照してください。

740C コンパイラでプログラミングする場合には、「コンパイラ・プログラミング・ガイド」を参照してください。

本書について

本ガイド書は、次の章から構成されています。

740 アセンブラ

「第 1 章 チュートリアル」では、740 アセンブラの最も重要な機能を使用して簡単な 740 機械コード・プログラムを作成する方法を説明します。また、XLINK や XLIB を使用した標準的な開発サイクルについても解説しています。

「第 2 章 アセンブラ・オプションの概要」では、740 アセンブラ・オプションの設定方法を解説し、アセンブラ・オプションをアルファベット順に示します。

「第 3 章 アセンブラ・オプションのリファレンス」では、各アセンブラ・オプションの内容を詳細に説明します。

「第 4 章 アセンブラ・ファイル形式」では、740 アセンブラのソース形式とアセンブラ・リストの形式を説明します。

「第 5 章 アセンブラ演算子の概要」では、アセンブラ演算子を優先順位で配列して示します。

「第 6 章 アセンブラ演算子のリファレンス」では 740 アセンブラ演算子をアルファベット順にリストし、各演算子を詳細に説明します。

「第 7 章 アセンブラ疑似命令のリファレンス」では、740 アセンブラ疑似命令をその関数に従ってグループに分類し、完全なリファレンス情報を提供します。

「第 8 章 アセンブラ命令」では 740 命令ニモニックをリストし、各ニモニックで使用可能なアドレス指定モードを詳細に説明します。

XLINK リンカ

「第 9 章 XLINK リンカ」では XLINK リンカを紹介し、XLINK リスト形式を説明します。

「第 10 章 XLINK オプションのリファレンス」では、XLINK オプションの設定方法を説明し、XLINK オプションをアルファベット順に示します。さらに XLINK オプションの内容を詳細に示します。

「第 11 章 XLINK 出力形式」では、XLINK で利用できる出力形式を要約します。

XLIB ライブラリアン

「第 12 章 XLIB ライブラリアン」では、再配置可能なルーチンのライブラリを生成・維持できるように作成された XLIB ライブラリアンを紹介します。

「第 13 章 XLIB コマンドの概要」では、XLIB コマンドの概要を説明します。

「第 14 章 XLIB コマンドのリファレンス」では、各 XLIB コマンドについての完全なリファレンス情報を提供します。

診断

「第 15 章 アセンブラ診断」では、740 アセンブラに固有なエラー・メッセージの一覧を提供します。

「第 16 章 XLINK 診断」、「第 17 章 XLIB 診断」では、XLINK や XLIB が発生するエラーや警告メッセージを説明し、各場面に応じた処置を示します。

前提条件

本ガイド書では、既に次の項目の操作知識があるものと想定して説明が行われます。

- 740 プロセッサ
- 740 アセンブラ言語
- ホスト・システムに応じた Windows

表記規則

本書では、次の表示規則を使用します。

形式	目的
`computer`	入力する、あるいは画面に表示されるテキストです。
`parameter`	コマンドの一部として入力しなければならない実際の値を表したラベルです。
`[option]`	コマンドの任意指定の部分です。
`{a | b | c}`	コマンドの選択肢です。
参照	本ユーザ・ガイドの別の箇所、あるいは別のガイド書の参照を意味します。
組み込みワークベンチ用	組み込みワークベンチのインターフェースに固有な命令です。
コマンドライン用	コマンドライン版に固有な命令です。
共通用	上記の **2** つのツールに共通な操作です。

本書中で、

- **SW とは、Spy Workbench の略で、シミュレータデバッガ C-SPY の Windows 95/NT 環境対応版です。**
- **その他、特に「コマンドライン版」との断わりがない「C-SPY」は、上記の「SW」のことを指します。**

目次

第１章　チュートリアル

このチュートリアルでは、EW 740 のアセンブラを使用して 740 プロセッサ用に一連の簡単な機械コード・プログラムを作成する方法を説明します。

この章を読む前に以下の準備が必要です。

- 製品付属の文書やテキストファイルなどを参照して、EW 740 をインストールしてください。
- 740 プロセッサのアーキテクチャ、命令セットに慣れておきます。詳細については、「アセンブラ命令」およびメーカーのデータ・ブックを参照してください。

サンプル・プログラムの実行

本チュートリアルでは別パッケージの SW（C-SPY）シミュレータをもっているものと想定して、SW を使用した例題プログラムの実行方法を説明します。

あるいは、情報をデバッグせずに例題プログラムをリンクすることによって実行し、**aout.a31** ファイルを与えることもできます。このファイルは、エミュレータや PROM プログラマにダウンロードされます。デフォルトのインテル標準形式以外のフォーマットを指定するには、XLINK の-F オプションを使用します。

スタート

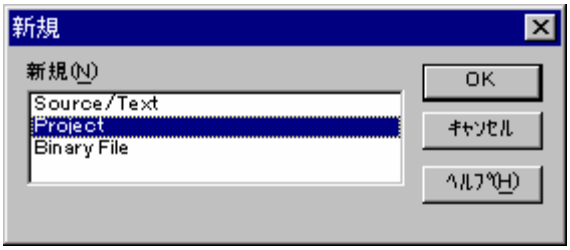
最初のステップは、チュートリアルプログラム用に新しいプロジェクトを生成することです。

新しいプロジェクトの生成

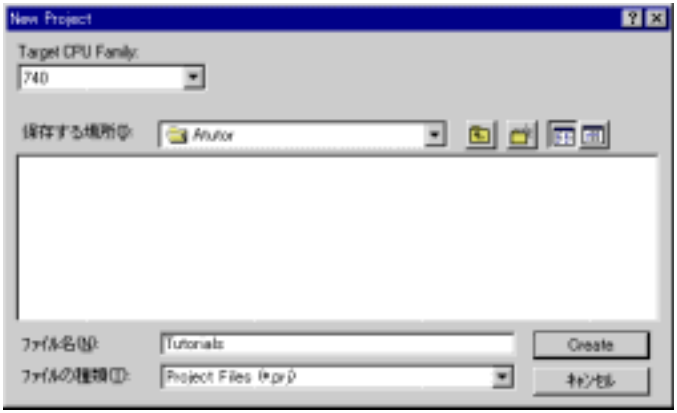
組み込みワークベンチ用　新しいプロジェクトの生成

まず、組み込みワークベンチを起動し、チュートリアル用のプロジェクトを次のように生成します。

File メニューより **New** を選択すると、次のようなダイアログボックスが表示されます。

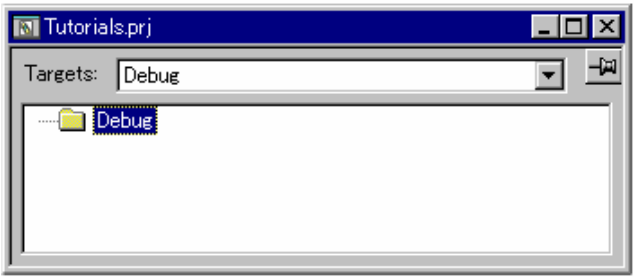


Project を選択し **OK** を選ぶと **New Project** のダイアログボックスが表示されます。**Tutorials** を**「ファイル名」**ボックスに入力し、**Target CPU Family** を **740** にセットします。



Create を選択すると、新しいプロジェクトが生成されます。

プロジェクト・ウインドウが表示されます。もし必要なら、**Targets** ドロップダウン・リストボックスより **Debug** を選択して、**Debug** ターゲットを表示ください。



次にチュートリアル・ファイルを含んだグループを、以下のように生成します。

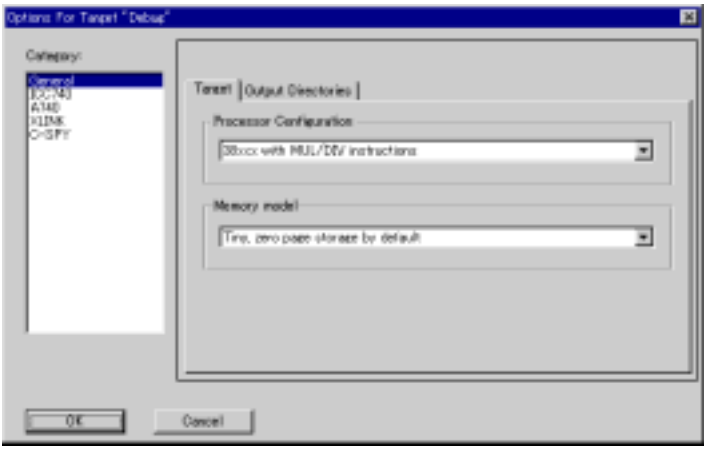
Project メニューから **New Group...**を選び、**Common Sources** という名前を入力します。デフォルトでは、両方のターゲットが選択されていますので、このグループも両方のターゲットが追加されます。

OK を選んでグループを生成します。プロジェクト・ウインドウが表示されます。

さらに、プロセッサとメモリモデルにあったターゲットオプションを設定します。

プロジェクト・ウインドウ内の **Debug** フォルダアイコンを選び、**Project** メニューより **Options...**を選び、**Category** リストの中から **General** を選び、**Target** タブをクリックして、ターゲットオプション・ページを表示します。

Processor Configuration を **38XXX with MUL and DIV instructions** にセットします。



OK を選択してターゲットオプションを保存してください。

コマンドライン用新しいプロジェクトの生成

特定のプロジェクトのファイルはすべて、他のプロジェクトやシステム・ファイルとは別に、1 つのディレクトリに格納することを勧めます。

チュートリアル・ファイルは、**A740** ディレクトリにインストールされます。次のようにコマンドを入力して、これらのファイルを作業ディレクトリをコピーします。

```
cd c:\iar\ew23\740
md TEST
cd TEST
copy ..\a740\*.*
```

このチュートリアルを使用する間、操作はこのディレクトリで行われます。したがって、生成されるファイルは、このディレクトリに保存されます。

チュートリアル 1

簡単なプログラムのアセンブルと実行

最初の例題プログラムは、アセンブラ・プログラムの書込み方法、およびそのプログラムのアセンブル、リンク、実行の方法を説明しています。

プログラムの書込み

最初のプログラム例は、A レジスタ、X レジスタを 2 進化 10 進法でカウントするループです。

```
          NAME      first

          ORG       $FFFC
reset     WORD      main        ; Reset vector

          ORG       $2000
main      SED                   ; Select decimal mode
          CLT                   ; Acc, not memory
          LDA       #0          ; Initialize low and
          TAX                   ; high byte of count
          CLC
loop      ADC       #1          ; increment low byte
          TAY
          TXA
          ADC       #0          ; carry to high byte
          TAX
          TYA
          BCC       loop
done      STP                   ; finished - do nothing
          BRA       done
          END
```

WORD 疑似命令は、main プログラムの開始アドレスをリセット・アドレスに置きます。

組み込みワークベンチ用プログラムの書込み

組み込みワークベンチ用を起動し、**File** メニューより **New** を選択して、**New** ダイアログボックスを表示します。

Source/Text を選び、**OK** を選択して新しいテキスト文書を開きます。

上記のプログラムを入力し、これをファイル **first.s31** として保存します。740 アセンブラに関連したファイルは、これらを識別するために**.s31**、**.a31**、**.d31** および**.r31** といった拡張子をもちます。

入力する代わりにアセンブラ・ファイル・ディレクトリに提供されているプログラムをコピーして用いることもできます。

コマンドライン用プログラムの書込み

標準のテキスト・エディタ（MS-DOS **EDIT** エディタなど）を使用して、上記のプログラムを入力し、これをファイル **first.s31** として保存します。

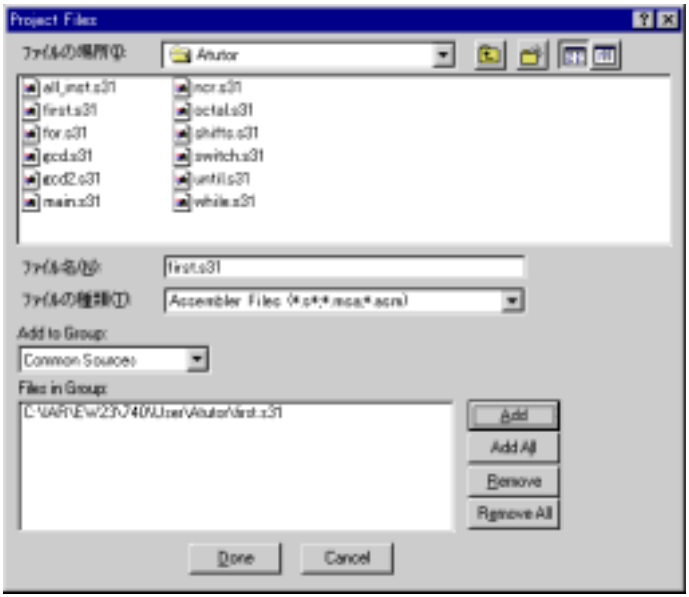
入力する代わりにアセンブラ・ファイル・ディレクトリに提供されているプログラムをコピーして用いることもできます。

プログラムのアセンブル

組み込みワークベンチ用プログラムのアセンブル

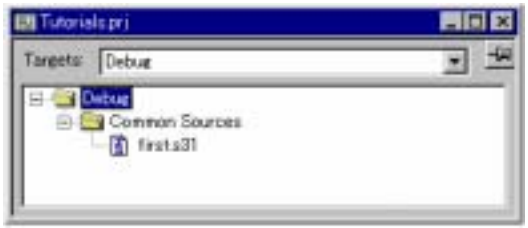
プログラムをアセンブルするには、まずこれを次のように Tutorials プロジェクトに追加します。

Project メニューより **Files...**を選び、**Project Files** ダイアログボックスを表示させます。ダイアログボックスのファイル選択リストに **first.s31** を配置し、Add を選択して、**Common Sources** グループにこれを追加します。



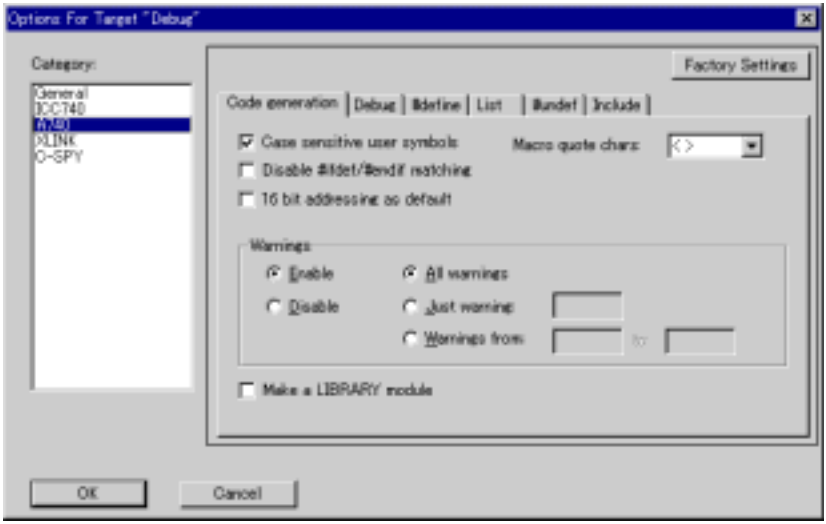
次に **Done** をクリックして、**Project Files** ダイアログボックスを閉じます。

+ シンボルをクリックして、プロジェクトウインドウ・ツリー内のファイルを以下のように表示させます。

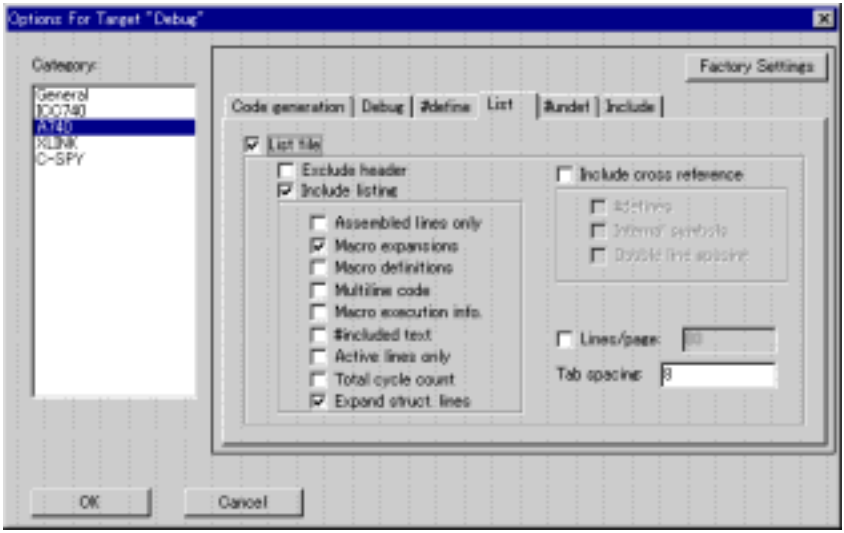


そして、次のようにプロジェクト用のアセンブラ・オプションを設定します。

プロジェクト・ウインドウの **Debug** フォルダアイコンを選び、**Project** メニューから **Options...** を選び、さらに **Category** リストの中から **A740** を選んでアセンブラ・オプションのページを表示させます。

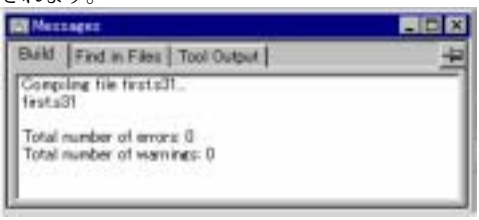


List をクリックしてリストオプションのページを表示させ、List file を選択してアセンブラリストファイルを生成するようにします。



OK を選択して、**Options** ダイアログボックスを閉じます。

ファイルをアセンブルするには、プロジェクト・ウインドウでそれを選択し、**Project** メニューより **Compile** を選んでください。進行状況が、メッセージウインドウに以下のように表示されます。



リストファイルが、**first.lst** の名前で **General** オプションページで指定されたフォルダに生成されます。デフォルトのフォルダは、**Debug\list** になります。**File** メニューの **Open...**を選択し、さらに適切なフォルダから **tutor1.lst** を選択してこのファイルを開いてみてください。

コマンドライン用プログラムのアセンブル

ファイルをアセンブルするには、プロンプトで次のコマンドを入力します。

```
a740 first -L
```

これによって、リストが **first.lst** ファイルに送られます。

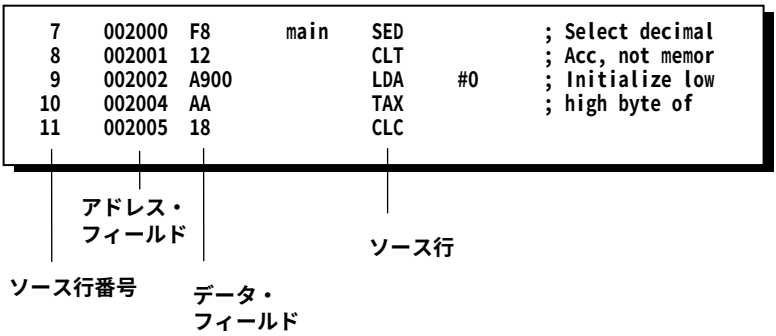
共通用リストの表示

リストファイルを見ると、次の内容が格納されているのが分かります。

```
#############################################################################
#
#     IAR Systems 740 Assembler V2.11A/WIN 15/Feb/2000  15:51:11
#
#          Target option =  740with MUL/DIV
#          Source file   =  first.s31
#          List file     =  first.lst
#          Object file   =  first.r31
#          Command line  =  first －L
#
#############################################################################

      1    00FFFC                    NAME    first
      2    00FFFC
      3    00FFFC                    ORG     $FFFC
      4    00FFFC 0020      reset    WORD    main    ; Reset vector
      5    00FFFE
      6    002000                    ORG     $2000
      7    002000 F8        main     SED             ; Select decimal mode
      8    002001 12                 CLT             ; Acc, not memory
      9    002002 A900               LDA     #0      ; Initialize low and
     10    002004 AA                 TAX             ; high byte of count
     11    002005 18                 CLC
```

```
    12    002006 6901     loop    ADC     #1      ; increment low byte
    13    002008 A8               TAY
    14    002009 8A               TXA
    15    00200A 6900             ADC     #0      ; carry to high byte
    16    00200C AA               TAX
    17    00200D 98               TYA
    18    00200E 90F6             BCC     loop
    19    002010 42       done    STP             ; finished - do nothing
    20    002011 80FD             BRA     done
    21    002013                  END
################################
#          CRC:6085            #
#        Errors:   0           #
#        Warnings: 0           #
#         Bytes: 21            #
################################
```

これは、各ソース・コード文によって生成された機械コード命令を示しています。
CRC 番号はアセンブリの日付によって異なり、変動することに注意してください。
リストの形式は、次のとおりです。



ソースのアセンブルが完了したと想定すると、リンク可能なオブジェクト・コードを格納した **first.r31** ファイルもまた作成されます。

プログラムの入力時にエラーがあった場合は、アセンブリの際にエラーが画面に表示されます。この場合は、エディタに戻り、ソース・コードを注意深く調べて全ての誤りを見つけ出し、修正し、同じファイル名でソース・ファイルを保存し直し、再度アセンブルを試みます。

プログラムのリンク

組み込みワークベンチ用プログラムのリンク

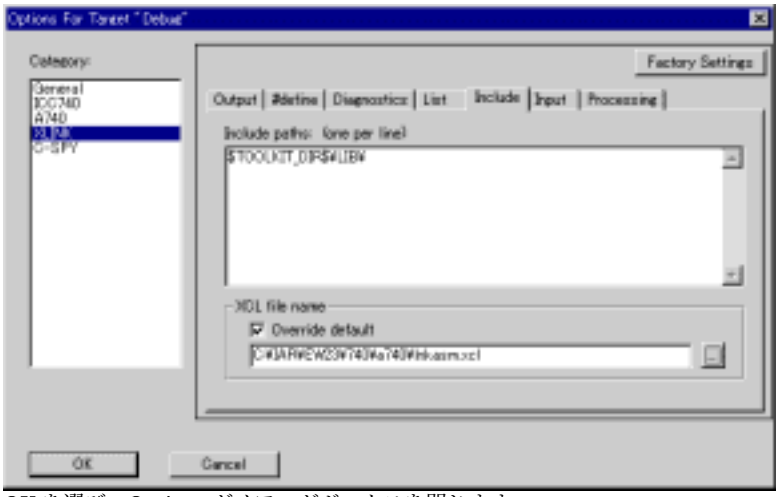
プログラムをリンクする前に、プロジェクトに対するリンカオプションの設定が必要です。

プロジェクト・ウインドウから **Debug** フォルダアイコンを選び、**Project** メニューから

Options...を選び、**Category** リスト内の **XLINK** を選択して XLINK オプションのページを表示させます。

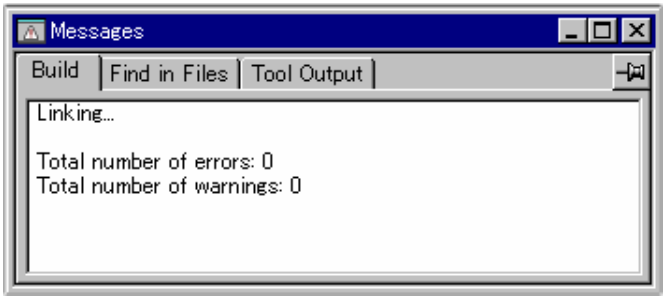
次に **Output** をクリックして、出力オプションを表示させます。SW (C-SPY)でのデバッグ用ファイルを生成するために、**Format** オプションが **Debug info with terminal I/O** にセットされているか調べてください。

さらに **Include** ページを表示させ、**XCL file name** の **Override default** をチェックして、デフォルトの XCL ファイル名に上書きします。...をクリックして、ファイルを参照できるようにします。**a740** ディレクトリより **lnkasm.xcl** を探して、選択します。このリンカ・コマンド・ファイルは、アセンブラファイルを使用するために特別に設計されています。



OK を選び、**Options** ダイアログボックスを閉じます。

ファイルをリンクするには、**Project** メニューより **Link** を選択してください。リンク中の進行状況がメッセージウインドウに以下のように表示されます。



コマンドライン用プログラムのリンク

オブジェクト・ファイルをリンクし、C-SPY デバッガが実行することのできるコードを作成するには、次のコマンドを入力します。

```
xlink first –c740 –r –f lnkasm
```

-c オプションは目的のプロセッサを指定し、**-r** は SW で使用できるようにデバッグ情報が含まれなければならないことを指定し、**-f** オプションは **lnkasm.xcl** をリンカ・コマンド・ファイルに指定します（ここでは、**a740** ディレクトリにあるこのファイルが、カレントディレクトリにコピーされていると仮定しています）。

デフォルトでは、出力コードは **aout.d31** ファイルに置かれます。

プログラムの実行

組み込みワークベンチ用プログラムの実行

SW (C-SPY)デバッガを使用して例題プログラムを実行するには、**Project** メニューから **Debugger** を選びます。

SW ウインドウが表示され、次の警告メッセージがレポートウインドウに表示されます。

```
Download completed, 21 bytes loaded
Warning[12]: Exit label missing
```

これらの警告は無視してください。

Window メニューより **Register** を選んで、SW でレジスタ・ウインドウを開いてください。

そして、**Execute** メニューから **Step** を選ぶか、F2 を押してプログラムをステップし、A と X レジスタのカウント値を 2 進化 10 進法で見てください。

File メニューから **Exit** を選択して、SW を終了します。

チュートリアル 2

マクロの使用法

このチュートリアルでは、簡単なマクロの使用法を説明します。このサンプル・プログラムは再帰的なルーチンを定義して二項係数を計算します。

アセンブラのマクロ機能の完全な説明については、「マクロ処理疑似命令」を参照してください。

二項係数 ${}^{n}Cr$ は、一度に n から r 取られる異なった組み合わせの数を与えます。たとえば文字 X、Y および Z から一度に 2 つ取る組み合わせは 3C2 または 3、すなわち XY、YZ、および XZ です。この関数は以下のように再帰的に定義することができます。

${}^{n}C_{r}$ = 1　　（r=0 のとき）
= 1　　（n=r のとき）
= ${}^{n-r}C_{r} + {}^{n-1}C_{r-1}$　（他の場合．ここで n は r より大きくなければならない）

このプログラムは、以下の簡単なマクロ **savexy** を定義し、X、Y レジスタを保存した状態でサブルーチンを呼び出します。

```
savexy    macro     routine
          TXA
          PHA
          TYA
          PHA
          JSR       routine
          STA       temp
          PLA
          TAY
          PLA
          TAX
          LDA       temp
          ENDM
```

このマクロは、次のような文で呼び出されます。

savexy ncr

この呼び出しはマクロ本体に展開され、**routine** が **ncr** で置換されます。

アセンブラ・プログラム **ncr** の完全なリストは、以下のとおりです。

```
savexy    macro     routine
          TXA
          PHA
          TYA
          PHA
          JSR       routine
          STA       temp
```

```
          PLA
          TAY
          PLA
          TAX
          LDA       temp
          ENDM

          ORG       $80
temp      BLKB      1           ; temp storage for ncr

          ORG       $FFFE
reset     WORD      main

          ORG       $2000
ncr       CPY       #0
          BEQ       one
          STY       temp
          CPX       temp
          BNE       none
one       LDA       #1
          RTS

none      DEX
          savexy    ncr         ; form n-1 C r
          PHA                   ; and save it
          DEY
          JSR       ncr         ; form n-1 C r-1
          STA       temp
          PLA
          CLC
          ADC       temp
          RTS

main      CLD                   ; Select binary mode
          CLT                   ; Acc, not memory
          LDX       #$FF
          TXS                   ; initialize stack

          LDX       #7
          LDY       #3          ; calculate 7 C 3
          JSR       ncr

done      STP                   ; finished - do nothing
          BRA       done
          END
```

メイン・プログラム **main** はスタックを設定し、X=7, Y=3 をもつ **ncr** サブルーチンを呼び出し、${}_7C_3$ を計算します。

ncr サブルーチンは、ルーチンが再帰的に再入力されたときにスタックを使用して中間結果を保存します。このサブルーチンは、それぞれ **n** と **r** の値を格納した X レジスタ、Y レジスタを伴って呼び出され、結果は A レジスタに返されます。

このプログラムを入力して **ncr.s31** というファイルに保存します。あるいは、アセンブラ・ファイル・ディレクトリ **a740** にあるソース・ファイルのコピーを使用します。

プログラムのアセンブル

組み込みワークベンチ用 プログラムのアセンブル

Tutorial プロジェクトを閉じて、**File** メニューから **New** を選び、新しいプロジェクト **Tutor2** を生成し、ファイル **ncr.s31** を追加します。以前のようにプロジェクト・ウインドウでこのファイルを選択し、**Project** メニューから **Compile** を選んで、アセンブルを実行します。

コマンドライン用プログラムのアセンブル

プログラムをアセンブルするには、次のコマンドを入力します。

```
a740 ncr -r –L
```

共通用 リストの表示

ncr.lst ファイルには、次の出力が生成されます。これ以降のリストでは、ヘッダー情報を省略して見やすくしています。

```
15    000080                ORG     $80
16    000080          temp  BLKB    1       ; temp storage for
                                               ncr
17    000081
18    00FFFE                ORG     $FFFE
19    00FFFE 2920     reset WORD    main
20    010000
21    002000                ORG     $2000
22    002000 C000     ncr   CPY     #0
23    002002 F006           BEQ     one
24    002004 8480           STY     temp
25    002006 E480           CPX     temp
26    002008 D003           BNE     none
27    00200A A901     one   LDA     #1
28    00200C 60             RTS
29    00200D
30    00200D CA       none  DEX
31    00200E                savexy  ncr     ; form n-1 C r
31.1  00200E 8A             TXA
31.2  00200F 48             PHA
31.3  002010 98             TYA
31.4  002011 48             PHA
31.5  002012 200020         JSR     ncr
31.6  002015 8580           STA     temp
31.7  002017 68             PLA
31.8  002018 A8             TAY
31.9  002019 68             PLA
31.10 00201A AA             TAX
31.11 00201B A580           LDA     temp
31.12 00201D                ENDM
```

```
32    00201D 48              PHA             ; and save it
33    00201E 88              DEY
34    00201F 200020          JSR     ncr     ; form n-1 C r-1
35    002022 8580            STA     temp
36    002024 68              PLA
37    002025 18              CLC
38    002026 6580            ADC     temp
39    002028 60              RTS
40    002029
41    002029 D8        main  CLD             ; Select binary mode
42    00202A 12              CLT             ; Acc, not memory
43    00202B A2FF            LDX     #$FF
44    00202D 9A              TXS             ; initialize stack
45    00202E
46    00202E A207            LDX     #7
47    002030 A003            LDY     #3      ; calculate 7 C 3
48    002032 200020          JSR     ncr
49    002035
50    002035 42        done  STP             ; finished - do nothing
51    002036 80FD            BRA     done
52    002038
53    002038                 END
```

マクロで生成された行には、31.1 や 31.2 のように行番号欄に.（ドット）をつけて識別しています。

プログラムのリンク

プログラムを実行できるようにするためには、アセンブラによって生成された再配置可能なファイルを全アドレスが決定されたオブジェクト・コードに変換する必要があります。

組み込みワークベンチ用プログラムのリンク

Project メニューより **Link** を選び、ファイルをリンクします。

コマンドライン用 プログラムのリンク

XLINK を実行して、次のコマンドでデバッグ用のコードを生成します。

```
xlink ncr –c740 -r –f lnkasm
```

ここで、リンクファイル **a740.xcl** は **a740** からカレントディレクトリにコピーされていると仮定しています。これによって、**aout.d31** ファイルが生成されます。

プログラムの実行

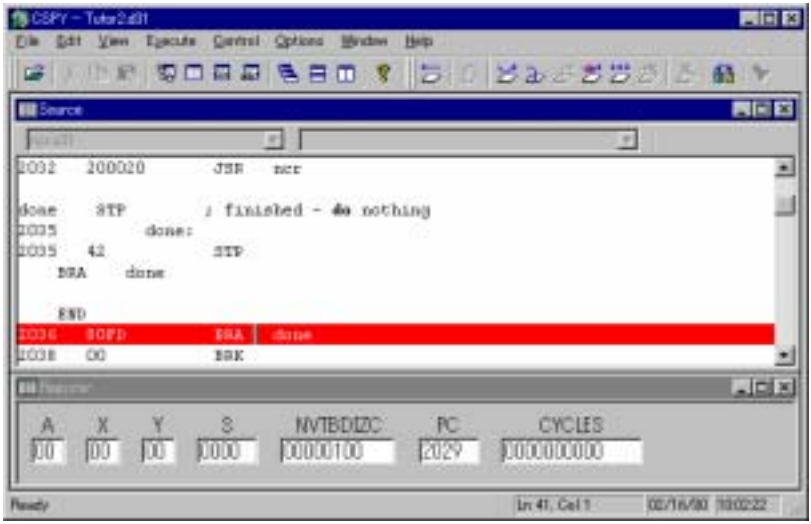
組み込みワークベンチ用プログラムの実行

SW デバッガを使用して例題プログラムを実行するには、以前のように **Project** メニューから **Debugger** を選び、警告メッセージは無視します。SW ウインドウが表示されます。

Execute メニューから **Step** を選び、ソース・ウインドウにソース・プログラムを表示させます。

次にソース・ウインドウ上で **main** プログラムの終わりの **BRA　done** 命令の行にカーソルを置き、**Control** メニューから **Toggle Breakpoint** を選んで、この位置にブレークポイントをセットします。

BRA　done 命令が強調表示され、ブレークポイントがここにセットされたことを示します。



Window メニューより **Register** を選んでレジスタ・ウインドウを開きます。そして **Execute** メニューから **Go** を選んで **main** からブレークポイントまで実行します。

結果が A レジスタに返され、値が 23（16 進）になります。

チュートリアル　3

モジュールの使用法

最後の例題は、ライブラリ・モジュールを生成し、XLIB ライブラリアンを使用してモジュールのファイルを管理する方法を説明します。

ライブラリの使用法

大型のプロジェクトを扱っている場合は、いくつかのプログラムで使用される一連の有用なルーチンをすぐに集めることができます。

ルーチンが必要になるたびにルーチンのアセンブルを回避するために、このルーチンをオブジェクト・ファイルとして保存することができます。すなわち、アセンブルするが、リンクさせるのではありません。

１つのオブジェクト・ファイルにある一連のルーチンをライブラリと言います。グラフィカ

ル・ライブラリや数学ライブラリなどの関連ルーチンを作成するには、ライブラリ・ファイルを使用することを勧めます。

XLIB ライブラリアンを使用してライブラリを操作することができます。これによって、次のことが行えます。

- **PROGRAM** 型から **LIBRARY** 型へ、あるいはその逆へモジュールを変更できる。
- モジュールをライブラリ・ファイルへ追加したり、あるいはライブラリ・ファイルから削除したりできる。
- エントリ名を変更できる。
- モジュール名、エントリ名などをリストできる。

メイン・プログラムの作成

メイン・プログラムは、単に **rightshift** というルーチンを使用して A レジスタの内容を右側にシフトします。A レジスタは start の値（この例では$AB）に設定され、**rightshift** ルーチンが呼び出されて、X レジスタの内容で指定されるように、値を 4 桁右側へシフトします。

```
            NAME        main

            EXTERN      rightshift

            RSEG        CODE
main        LDA         start
            LDX         #4
            JSR         rightshift
            RTS

            RSEG        DATA
start       BYTE        $AB

            END         main
```

EXTERN 疑似命令は **rightshift** がリンク時に決定されるように、これを外部シンボルとして宣言します。

これを入力し、**main.s31** ファイルとして保存します。あるいは、アセンブラ・ファイル・ディレクトリ **a740** にあるファイルをコピーします。

ライブラリ・ルーチンの作成

2 番目のプログラムは、個々にアセンブルされたライブラリの形成に使用されます。これは、**main** で呼び出される **rightshift** ルーチンと、それに対応した **leftshift** ルーチンの 2 つのライブラリ・ルーチンもっています。これらのルーチンは両方とも、A レジスタの内容に作用し、内容を繰り返し右や左にシフトさせます。シフトの実行回数は、X レジスタをゼロにデクリメントさせることによって制御されます。

```
          MODULE     rightshift
          PUBLIC     rightshift
          RSEG       CODE
rightshift
          LSR        A
          DEX
          BNE        rightshift
          RTS
          ENDMOD

          MODULE     leftshift
          PUBLIC     leftshift
          RSEG       CODE
leftshift
          ASL        A
          DEX
          BNE        leftshift
          RTS

          END
```

このルーチンは、**MODULE** 疑似命令でライブラリ・モジュールとして定義されます。このライブラリ・モジュールは、別のモジュールによって呼び出された場合に限ってこれらを含むように XLINK リンカに指示します。

rightshift および **leftshift** のエントリ・アドレスは、**PUBLIC** 疑似命令によってその他のモジュールに対して公開されます。

これらのモジュールを **shifts.s31** というソース・ファイルに保存します。あるいは、アセンブラ・ファイル・ディレクトリ **a740** にあるファイルをコピーします。

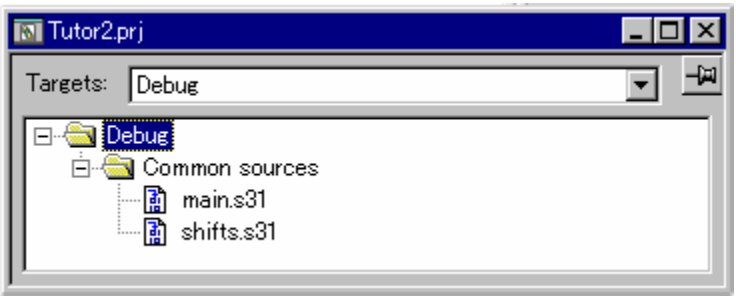
ソース・ファイルのアセンブルとリンク

次に、上記の両ソース・ファイルをアセンブルする必要があります。

両方のソース・ファイルをいっしょにアセンブルできるとはいえ、大きなプロジェクトでは、このことはかなりの時間を消費します。ライブラリ・ルーチンを別々にアセンブルすることにより、メイン・プログラムの変更は、メイン・ソース・ファイルの再アセンブルを必要とするだけになります。

組み込みワークベンチ用 **アセンブルとリンク**

以前のチュートリアルで説明したように、**main.s31** と **shifts.s31** を含むプロジェクトを生成します。



両方のファイルをアセンブル、リンクするには、**Project** メニューより **Make** を選択します。

コマンドライン用アセンブルとリンク

メイン・プログラムをアセンブルするには、次の入力を行います。

```
a740 main -r -L
```

同じように、ライブラリ・ルーチンをアセンブルするには、次の入力を行います。

```
a740 shifts -r -L
```

これらのソース・ファイルをアセンブルすると、2つの再配置可能なファイルが作成されます。これらのファイルをリンクし、メイン・プログラムおよびメイン・プログラムが参照するライブラリ・ルーチンをもった単一の実行可能なオブジェクト・ファイルを、全ての相互参照を決定させておいて作成することが必要です。この場合、あるセクションから別のセクションへの参照だけが **rightshift** サブルーチンの呼び出しとなります。**leftshift** ルーチンは、まったく使用されません。

ファイルをシングル・ステップでリンクするには、コマンド行から（1行で）次のように入力します。

```
xlink main shifts –c740 -Z(CODE)CODE=2000 –Z(DATA)DATA=80 -xsm -l main.map
```

下表に、コードとデータ・セグメントのアドレスを定義するオプションを説明します。

パラメータ	内容
-Z(CODE)CODE=2000	コード・セグメントを 16 進アドレス 2000 に再配置されるように指定する。
-Z(DATA)DATA=80	データ・セグメントを 16 進アドレス 80 に再配置されるように指定する。
-xsm	セグメント、モジュール・マップをもつ相互参照リストを要求する。
-l main.map	リストの出力を **main.map** に指定する。

XLINK オプションに関する詳細内容については、「XLINK オプションのリファレンス」を参照してください。

共通用 リストの表示

main.map 相互参照リスト（クロス・リファレンス）をリストさせると、XLINK で作成されたモジュールには **main** プログラム・モジュールと **rightshift** ライブラリ・モジュールは含まれ、未使用の **leftshift** ライブラリ・モジュールは含まれていないのが確認できます。

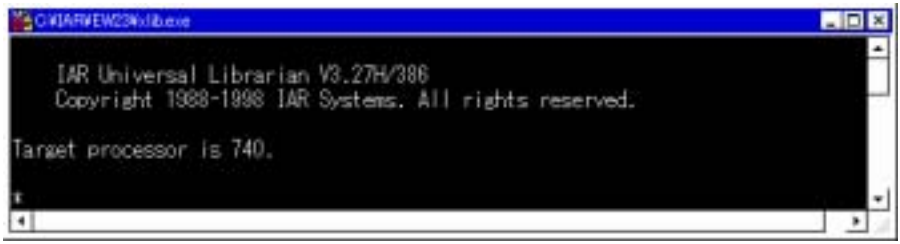
XLIB ライブラリアンの使用法

上記で説明した **rightshift** モジュールや **leftshift** モジュールのような汎用向けモジュールのアセンブル、デバッグを完了したら、XLIB ライブラリアンを使用してモジュールをライブラリに追加します。

組み込みワークベンチ用 XLIB ライブラリアンの実行

Project メニューより **Librarian** を選び、XLIB ライブラリアンを実行します。

XLIB ウインドウが表示されます。



これで、***** プロンプトの位置に XLIB コマンドを入力可能になります。

コマンドライン用 XLIB ライブラリアンの実行

次の入力を行って、XLIB ライブラリアンを起動します。

```
XLIB
```

XLIB ライブラリアンは対話型モードで実行され、コマンドを入力するための ***** プロンプトを表示します。

共通用 XLIB コマンドの指定

希望するモジュールを **shifts.r31** ファイルから **math.r31** というライブラリに取り出します。これを行うには、次のコマンドを入力します。

```
FETCH-MODULES
```

これによって、次の引数の入力が求められます。

プロンプト	入力項目
Source file	**shifts** ↵
Destination file	**math** ↵
Start module	↵（ファイルの最初の項目であるデフォルト値を使用する）
End module	↵（ファイルの最後の項目であるデフォルト値を使用する）

これによって、**leftshift** ルーチン、**rightshift** ルーチン用のコードをもった **math.r31** ファイルが作成されます。

これは、次の入力を行って確認できます。

LIST-MODULES

すると、次の引数の入力が求められます。

プロンプト	入力項目
Object file	**math**
List file	↵（画面の使用）
Start module	↵（最初のモジュールからの開始）
End module	↵（最後のモジュールで終了）

最後に、

EXIT ↵

と入力してライブラリアンを終了します。

さらにモジュールを **math** ライブラリに追加する際には、いつでも同じ手順を用いることができます。

第２章　アセンブラ・オプションの概要

この章では、アセンブラ・オプションの要約をアルファベット順に紹介し、組み込みワークベンチまたはコマンドラインからのオプションの設定方法を説明します。

このオプションは、組み込みワークベンチの **A740** と **General** オプションのページに対応して、以下の節に分けられます。

Code generation	List	Target
Debug	#undef	
#define	Include	

各オプションのリファレンス情報の詳細については、次章を参照してください。

アセンブラ・オプションの設定

組み込みワークベンチ用アセンブラ・オプションの設定

組み込みワークベンチでアセンブラ・オプションを設定するには、**Project** メニューより **Options...**を選び、**Category** リストから **A740** を選んでアセンブラ・オプションのページを表示させます。

次に表示または変更させたいオプションのカテゴリに対応するタブをクリックします。

すべての設定をデフォルトの設定値に戻すには、**Factory settings** ボタンをクリックしてください。

コマンドライン用 アセンブラ・オプションの設定

コマンドラインでアセンブラ・オプションを設定するには、a740 コマンドの後にそれらをコマンドラインに指定します。たとえば、ソース **first** をアセンブルして、デフォルトのリスト・ファイル **first.lst** を生成するには、次のように入力します。

```
a740 first -L
```

オプションの中には、間にスペースをはさんでファイル名を受け付けるものもあります。たちえば、ファイル **list.lst** でリストを生成するには、以下のように入力します。

```
a740 first -l list.lst
```

他のいくつかのオプションは、ファイル名ではない文字列を受け付けます。この文字列は、間にスペースをはさまないで、オプションに続きます。たとえば、デフォルトのファイル名リストをサブディレクトリに生成するには、以下のように入力します。

```
a740 first -Llist
```

アセンブラ・オプションの概要

下表に、全てのアセンブラ・オプションを示します。オプションの詳細内容については、次章「アセンブラ・オプションのリファレンス」でオプションの分類名で参照してください。

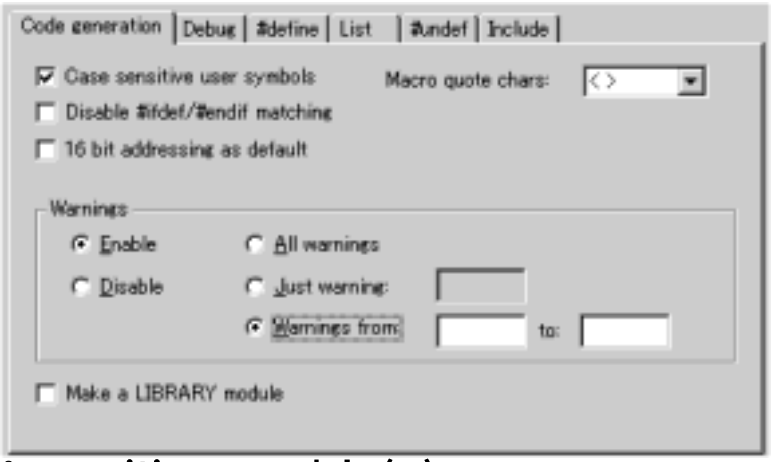
オプション	内容	分類
-B	マクロ実行情報を出力する。	List
-b	オブジェクトをライブラリ・モジュールにする。	Code generation
-c{DMEAO}	リスト・オプションを設定する。	List
-D*symb*[=*xx*]	オプション値でシンボルを定義する。	#define
-d	**#ifdef / #endif** がそろっているかをチェックしない。	Code generation
-E*number*	エラーの最大数を設定する。	コマンドライン
-f *filename*	コマンド行を拡張する。	コマンドライン
-G	標準入力をソースとして開く。	コマンドライン
-I*prefix*	インクルード検索接頭語を追加する。	Include
-i	インクルード・ファイルをリストする。	List
-L[*prefix*]	前置ソース名にリストを生成する。	List
-l *filename*	命名ファイルにリストを生成する。	List
-Mab	マクロ引数引用文字を設定する。	Code generation
-N	リストにヘッダーをつけません。	List
-O*prefix*	オブジェクト・ファイル名を設定する。	コマンドライン
-o *filename*	オブジェクト・ファイル名接頭語を設定する。	コマンドライン
-p*nn*	各ページの行数を設定する。	List
-r[en]	オブジェクトでデバッガ出力を使用可能にする。	Debug
-S	アセンブラのサイレント演算を設定する。	コマンドライン
-s{+|-}	ユーザー・シンボルの大文字／小文字の区別を設定する。	Code generation
-T	アクティブ行だけをリストする。	List
-t*n*	タブ・スペーシングを設定する。	List
-v*n*	プロセッサ・コンフィギュレーション	List
-U*symb*	シンボルを未定義にする。	#undef
-w[*string*]	警告を不能にする。	Code generation
-x{DI2}	相互参照リストを生成する。	List

第３章　アセンブラ・オプションのリファレンス

この章では、740 アセンブラ・オプションを機能分類項目に分類してそれぞれのオプションを詳細に説明します。

Code generation

これらのオプションは、アセンブラのコード生成を制御します。



Case sensitive user symbols (-s)

構文：　-s{+|-}

アセンブラがユーザーのシンボルの大文字／小文字を区別するかどうかは、次のように設定します。

コマンドライン	内容
-s-	オフ
-s+	オン

デフォルトでは、大文字／小文字の区別はオンになります。これは、例えば **LABEL** と **label** を異なったシンボルとみなすことを意味します。**-s-**オプションを使用すると、大文字・小文字の区別をオフにすることができます。この場合、**LABEL** と **label** は同じシンボルとみなされます。

Disable #ifdef/#endif matching (-d)

構文： -d

#ifdef…#endif がそろっていなくてもエラーを発生しないようにします。

#indef…#endif がそろっているかどうかのチェックはそれぞれのモジュールごとに行われ、それゆえ、モジュールの外の**#endif** は、通常エラーになります。このオプションを用いてチェックを中止します。

これを使うと、以下のような構文の記述が可能になります。

```
#ifdef     Version1
           MODULE M1
           NOP
           ENDMOD
#endif
           MODULE M2
           •
           •
           etc
```

Macro quote chars (-M)

構文： -M*ab*

各マクロ引数の左右の引用符に使用される文字をそれぞれ ***a***、***b***に設定します。

デフォルトの文字は<、>です。**-M** オプションを使用すると、代替規則に合わせて引用文字を変更したり、あるいは単にマクロ引数が< や> を含むようにすることができます。

組み込みワークベンチ用

ドロップ－ダウン・リストから 4 つのタイプの括弧から 1 つを選択できます。

コマンドライン用

例えば、ソース・ファイルに
-M[]
オプションを使用し、例えば、
print [>]
と書き込んで>を引数としてもつマクロ出力を呼び出します。

Warnings (-w)

構文： -w[*string*]

警告を不能にします。デフォルトでは、アセンブラはソース・ファイルの違法な要素を検出し、それがプログラミング・エラーによるものと考えられる場合は、警告メッセージを表示します（詳細については、「アセンブラの診断」を参照してください）。**-w** オプションに次の文字列をつけないで使用すると、全ての警告が不能になります。文字列をつけて**-w** オプションを使用すると、次のように作用します。

ｺﾏﾝﾄﾞﾗｲﾝ	効果	組み込みワークベンチ
+	全ての警告を可能にします。	**Enable & All warnings**
-	全ての警告を不能にします。	**Disable & All warnings**
+n	警告ｎだけを可能にします。	**Enable & Just warning**
-n	警告ｎだけを不能にします。	**Disable & Just warning**
+m-n	ｍ～ｎの警告を可能にします。	**Enable & Warnings from**
-m-n	ｍ～ｎの警告を不能にします。	**Disable & Warnings from**

Make a LIBRARY module (-b)

構文： -b

このオプションを使用すると、オブジェクト・ファイルがプログラム・モジュールではなくライブラリ・モジュールになります。

デフォルトでは、アセンブラは XLIB にいつでもリンク可能なプログラム・モジュールを生成します。アセンブラに、XLIB で使用するライブラリ・モジュールを生成させたい場合は、-b オプションを使用します。

（プログラム・モジュール名を指定するために）NAME 疑似命令をソースで使用する場合、-b オプションは無視されます。つまり、アセンブラは設定にかかわりなく、プログラム・モジュールを生成します。

Debug

Debug オプションは、オブジェクトのデバッグ情報の有無、およびその内容を制御します。

Generate debug information (-r)

構文： -r[n][e]

コンパイラに、SW(C-SPY)やその他のデバッガで要求される追加情報をオブジェクト・モジュールに入れるようにさせます。通常コンパイラはデバッギング情報を含みません。SW でコードのデバッギングを可能にするには、修飾子を付けないで **Generate debug information (-r)**オプションを使用し、オブジェクト・コード内にソース・ファイル参照を入れるようにします。完全なソース・ファイル情報をオブジェクト・コード内に含めるには、**e** 修飾子を使用します。

通常、**Generate debug information (-r)**オプションはデフォルトで **File reference** が選択されていて、外部のソース・ファイルを参照し、オブジェクト・ファイルにソースを含みません。**Embedded source (e)**を選択するとオブジェクトに完全なソース情報が含まれます。これを抑制させて、オブジェクト・ファイルのサイズを削減したい場合は、**Suppress source in object code (n)**を使用します。

#define

このオプションは、シンボルの定義を可能にします。

オプション値によるシンボルの定義（-D）

構文： -D*symb*[=*xx*]

名前 ***symb*** と値 ***xx*** でシンボルを定義します。値が指定されない場合は、１が使用されます。

-D オプションを使用すると、別の方法ではソース・ファイルで指定しなければならない値あ

るいは選択肢をより便利にコマンド行で指定することができます。例えば、シンボル **testver** が定義されたかどうかに応じてプログラムのテスト・バージョンまたは生産バージョンを作成するようにソース・ファイルをアレンジすることができます。これを行うには、次のようにインクルード・セクションを使用します。

```
#ifdef testver
...    ; テスト・バージョンだけの追加コード行
#endif
```

その後、コマンド行で要求されるバージョンを次のように選択します。

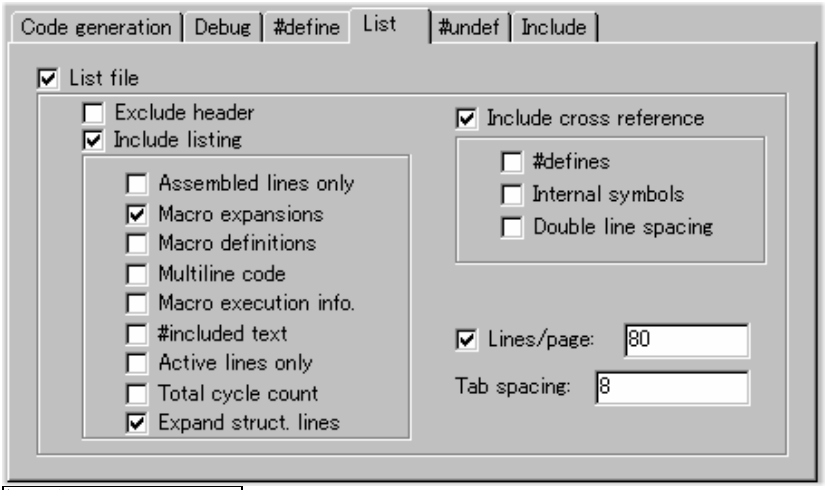
```
production version:  a740 prog
test version:        a740 prog -Dtestver
```

あるいは、ソース・ファイルが、しばしば変更する必要のある変数を使用することもあります。ソース・ファイルでは変数を未定義にしておき、**-D** オプションを使用してコマンド行で値を指定します。例えば、次のとおりです。

```
a740 prog -Dframerate=3
```

List

List オプションは、アセンブラにリストを生成させ、内容を選択し、その型を指定させます。



組み込みワークベンチ用 **List file**

アセンブラにリストを生成させ、ファイル ***sourcename*.lst** に送ります。

List file が選択されると次のリスト・オプションが利用できます。

オプション	説明
Include header	リストのヘッダを含む
Include listing	リストの本体を含む

Include listing を選択すると、次のオプションが利用できます。

オプション	説明
#included text	リストに**#include** ファイルを含む
Active lines only	リストにアクティブな行のみを含む
Macro definitions	リストにマクロ定義を含む
Macro expansions	リストにマクロ拡張を含む
Macro execution info	それぞれのマクロの呼び出しにマクロ実行情報をプリントする
Assembled lines only	アセンブルされた行のみをリストにする
Multiline code	必要な場合、複数行の命令で生成されたコードをリストにする

コマンドライン用 命名ファイルへのリストの生成（-l）

構文： **-l** ***filename***

このオプションを使用すると、アセンブラはリストを生成し、それを命名ファイルに送信します。拡張子を指定しない場合は**.lst** が使用されます。ファイル名の前に空白をいれてください。

デフォルトでは、アセンブラはリストを生成しません。**-l** オプションは作表をオンし、リストを特定のファイルに送ります。作表をオンし、デフォルトのファイル名に送信するには、代わりに**-L** オプションを使用します。

コマンドライン用 接頭語付きソース名へのリストの生成（-L）

構文： **-L**[***prefix***]

このオプションを使用すると、アセンブラはリストを生成し、それを ***prefixsourcefilename*.lst** ファイルに送信します。接頭語の前には空白をいれないでください。

デフォルトでは、アセンブラはリストを作成しません。単にリストを作成するには、接頭語をつけずに-L オプションを使用します。リストは、ソース・ファイルと同じ名前をもち拡張子**.lst** のあるファイルに送信されます。

-L オプションを使用すると、接頭語を指定して、例えば、次のようにリスト・ファイルをサブディレクトリに向けて送信することができます。

```
a740 prog –Llist\
```

これは、リストを、デフォルトの **prog.lst** ではなく **list\prog.lst** に送信します。

-L オプションは、**-l** と同時に使用することはできません。

コマンドライン用 リストにヘッダをつけない（-N）

構文： **-N**

通常リストに出力されるヘッダを不能にします。

#included text (-I)

構文： -i

このオプションを使用すると、**#include** ファイルがリストに組み込まれます。

デフォルトでは、アセンブラは**#include** ファイル行をリストしません。それは、これらの行は標準ファイルから取り込まれることが多く、リストのスペースを無駄にしてしまうからです。**-i** オプションを使用すると、必要に応じて**#include** ファイルをリストすることができます。

Active lines only (-T)

構文： -T

このオプションを使用すると、例えば偽の**#if** ブロックにある行ではなく実行可能状態の行のみをリストにいれることができます。デフォルトでは、全ての行がリストされます。

このオプションは、コードを生成しない、あるいはコードに影響を与えることのない行を除去することによってリストのサイズを削減する場合に有用です。

コマンドライン用 リスト・オプションの設定（-c）

構文： -c{DMEAO}

次の項目を１つ以上設定します。

オプション	内容
D	リスト作成を不能にします。
M	マクロ定義をリストします。
E	マクロ拡張を不能にします。
A	アセンブルされた部分のみをリストします。
O	複数行のコードをリストします。

Macro execution info (-B)

構文： -B

-B オプションを使用すると、アセンブラはマクロが呼び出されるたびにマクロ実行情報を標準出力ストリームに出力します。マクロ実行情報は、次のものから成ります。

- マクロ名
- マクロの定義
- マクロの引数
- マクロの拡張テキスト

Include cross reference (-X)

構文： -X{DI2}

このオプションを使用すると、アセンブラはリストの最後に相互参照リスト（クロス・リファ

レンス）を生成します。詳細については、「アセンブラ・ファイルの形式」を参照してください。

ｺﾏﾝﾄﾞﾗｲﾝ	組み込みワークベンチ
D	#define
I	Internal symbols
2	Dual line spacing

Lines/page (-p)

構文： -p*nn*

ページ毎の行数を *nn* に設定します。*nn* は、１０～150 の範囲でなければなりません。

Tab spacing (-t)

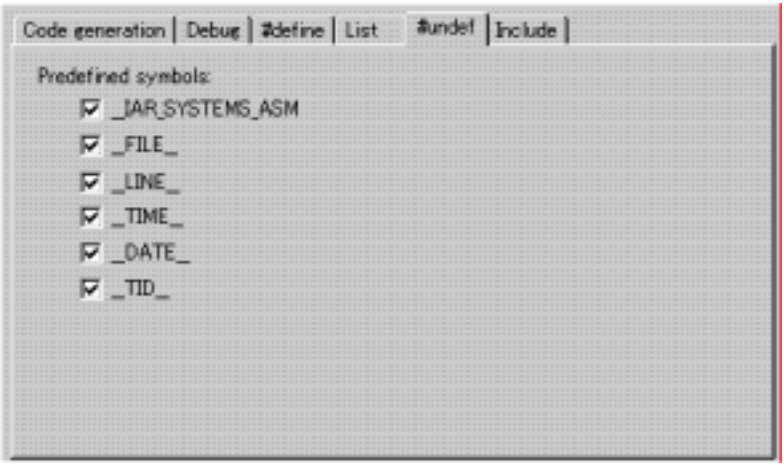
構文： -t*n*

タブ・ストップ毎の文字位置の数を *n* に設定します。*n* は、２～９の範囲でなければなりません。

デフォルトでは、アセンブラはタブ・ストップ毎に８文字位置を設定します。

#undef

#undef オプションは、以前に定義したシンボルを未定義にします。



シンボルの未定義（-U）

構文： -U*symb*

シンボル *symb* を未定義にします。

デフォルトでは、アセンブラはある定義済みシンボルを提供します。定義済みシンボルについては、40 ページの「定義済みシンボル」を参照してください。**-U** オプションを使用すると、このような定義済みシンボルを未定義にし、その名前を、以降の**-D**（シンボルの定義）オプションやソースの定義によって自分の使用に合わせて利用することができます。

組み込みワークベンチ用

シンボルを未定義にするには、これを **Predefined symbols** リストの中で、非選択にします。

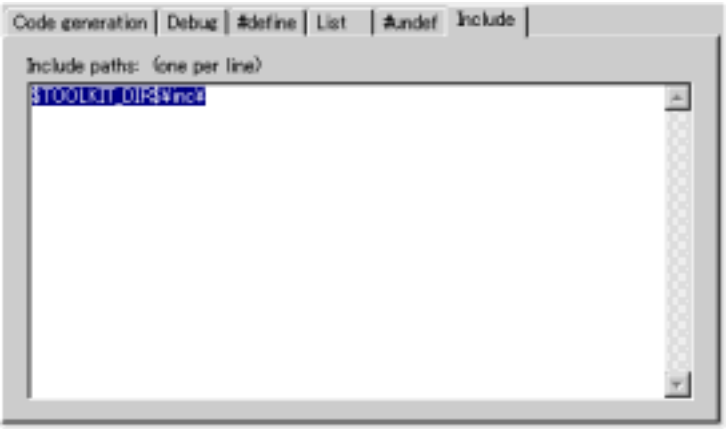
コマンドライン用

定義済みシンボル名 **__TIME__** を自分の目的に使用したい場合は、次のように定義済みシンボルを未定義にします。

```
a740 prog -U __TIME__
```

Include

Include オプションは、アセンブラ用にインクルード・パスを定義します。



#インクルード検索接頭語の追加（-I）

構文： **-I***prefix*

#include ファイル検索接頭語 ***prefix*** を追加します。

デフォルトでは、アセンブラはカレント作業ディレクトリのみで**#include** ファイルを検索しますが、**-I** オプションを使用すると、アセンブラに、カレント作業ディレクトリでファイルを見つけられなかった場合に検索を行うディレクトリ名を与えることができます。たとえば、次のオプションを使用し、

```
-Ic:\global\ -Ic:\thisproj\headers\
```

そして、ソースで次のように記述すると、

```
#include "asmlib.hdr"
```

アセンブラはまず **asmlib.hdr** ファイルを、次に **c:\global\asmlib.hdr** ファイルを、そして最後に **c:\thisproj\headers\asmlib.hdr** ファイルを検索します。

Target

General カテゴリの **Target** オプションは、アセンブラと C コンパイラ用にプロセッサとメモリ・モデルの指定を行います。これらのオプションの内容については、『コンパイラ・プログラミング・ガイド』を参照してください。

コマンドライン

以下の追加のオプションは、コマンドラインで使用できます。

-E*number*	エラーの最大数を設定する。
-f ***filename***	コマンド行を拡張する。
-G	標準入力をソースとして開く。
-O*prefix*	オブジェクト・ファイル名を設定する。
-o ***filename***	オブジェクト・ファイル名接頭語を設定する。
-S	アセンブラのサイレント操作を設定する。

コマンドライン用 最大エラー数の設定（-E）

構文： -E*number*

アセンブラから通知されるエラーの最大数を設定します。

デフォルトでは、最大エラー数は 100 となります。**-E** オプションを使用すると、例えば 1 アセンブリでより多くのエラーを確認するなど、この数値を増減させることができます。

コマンドライン用 コマンド行の拡張（-f）

構文： -f *filename*

***filename*.xcl** ファイルから読み込んだテキストによってコマンド行を拡張します。オプション自体とファイル名の間には空白をいれてください。

-f オプションは、コマンド行よりもファイルに置いた方がより便利なオプションが多数ある場合に特に有用です。例えば、**asmopt.xcl** ファイルから取り出されたオプションでアセンブラを実行するには、次のように使用することができます。

a740 prog -f asmopt

コマンドライン用 標準入力をソースとして開く（-G）

構文： **-G**

このオプションを使用すると、アセンブラは指定されたソース・ファイルではなく標準入力ストリームからソース・ファイルを読み取ります。

-G を使用すると、ソース・ファイル名を指定することはできません。

コマンドライン用 オブジェクト・ファイル名接頭語の設定（-O）

構文： **-O*prefix***

オブジェクトのファイル名に使用する接頭語を設定します。接頭語の前には空白をいれないでください。

デフォルトでは、接頭語は空白となります。したがって、（**-o** が使用されない限り）オブジェクト・ファイル名はソース・ファイル名と一致します。**-O** オプションを使用すると、次のように接頭語を指定して、例えばオブジェクト・ファイルをサブディレクトリに向けて送信することができます。

a740 prog –Oobj

これは、オブジェクトをデフォルトの **prog.r31** ファイルではなく、**obj\prog.r31** ファイルに送信します。

-O オプションは、**-o** オプションと同時に使用することはできません。

コマンドライン用 オブジェクト・ファイル名の設定（-o）

構文： **-o *filename***

オブジェクトに使用するファイル名を設定します。ファイル名の前には空白をいれてください。拡張子を指定しない場合は、**.r31** が使用されます。

デフォルトでは、アセンブラは拡張子を**.r31** に変更してソース・ファイル名を使用します。**-o** オプションを使用すると、オブジェクトに別のファイル名を使用することができます。例えば、

a740 prog -o obj

と入力すると、デフォルトの **prog.r31** ファイルではなく **obj.r31** ファイルにオブジェクトを格納します。

オプションとファイル名の間に空白をいれてください。

-o オプションは、**-O** オプションと同時に使用することはできません。

コマンドライン用 サイレント操作の設定（-S）

構文： -S

このオプションを使用すると、アセンブラは標準出力ストリームにメッセージを送信せずに実行されます。

デフォルトでは、アセンブラは標準出力ストリームを介してさまざまな重要でないメッセージも端末に送信します。**-S** オプションを使用すると、これを防止し、画面表示の混雑を減少させることができます。アセンブラはエラーや警告メッセージをエラー出力ストリームに送信しますので、このオプションの設定にかかわらず、これらは端末に表示されます。

第４章　アセンブラ・ファイル形式

740 アセンブラは、37XXX、38XXX グループから成る 740 ファミリ・マイクロプロセッサをサポートしています。

この章では、A740 アセンブラのソース形式とアセンブラ・リストの形式を説明します。

ソース形式

アセンブラ・ソース行の形式は、次のとおりです。

［ラベル［:］］演算［オペランド］[[;］コメント］

ここで、コンポーネントは次のとおりです。

ラベル　現在のロケーション・カウンタ(PLC)の値と型が割り付けられたラベルです。ラベルが最初のカラムから始まる場合、：（コロン）は任意指定です。

演算　アセンブラ命令あるいは疑似命令です。これは、最初のカラムからスタートしてはいけません。

オペランド　コンマで分離され、スペースやタブを含まない１つ以上のオペランド。

コメント　先頭に；（セミコロン）がつけられたコメント

各フィールドは、空白やタブで分離します。

ソース行は、255 文字を超えることはできません。

タブ文字（ASCII 09H）は、最も一般的な方法で拡張します。すなわち、８、１６、２４桁などに拡張できます。

式と演算子

式は、オペランドと演算子から成ります。

アセンブラは、算術演算と論理演算の両方を含む幅広い範囲の式を受け付けます。全ての演算子は３２ビットの２の補数整数を使用し、レンジのチェックは値がコード生成に使用された場合にのみ行われます。

式は、演算子の優先順位によって無効にならない限り左から右に評価されます。

式の中で有効なオペランドは、次のとおりです。

- ユーザ定義シンボルおよびラベル
- 浮動小数点定数を含まない定数
- ロケーション・カウンタ（PLC）シンボル＊

これらの項目は、次の項で詳細に説明します。

有効な演算子は、４９ページの第５章「アセンブラ演算子の概要」、５３ページの第６章「アセンブラ演算子のリファレンス」で説明します。

真と偽

式の中では、０の値は偽とみなされ、０以外の値は真と考えられます。

条件式では、偽の場合は０が、真の場合は１が返されます。

再配置可能な式でのシンボルの使用法

再配置可能なセグメントにシンボルを含む式は、XLINK によって位置決めされるセグメントの場所によって異なるため、アセンブリ時に決定することはできません。

このような式は、リンク時に XLINK によって評価・決定されます。式にはなんら制限はありませんので、演算子はあらゆるセグメント、あるいはセグメントの組合せから得られたシンボルで使用することができます。

例えば、プログラムは DATA, CODE セグメントを次のように定義することができます。

```
          EXTERN    third
          RSEG      DATA
first     BLKB      5
second    BLKB      3
          ENDMOD
          RSEG      CODE
start     -
```

そして、CODE セグメントでは次の命令が有効です。

```
          LDA       #first+7
          LDA       #first-7
          LDA       #7+first

          LDA       #(first/second)*third
```

シンボル

ユーザ定義シンボルの長さは最大 255 文字で、全ての文字は有効です。

シンボルは、a ～ z、A ～ Z、？（疑問符）、@、あるいは__（下線）文字から始めてください。シンボルには、0～9の数値、$（ドル記号）を含むことができます。ユーザ定義シンボルについては、大文字／小文字は有効です。命令、レジスタ、演算子、疑似命令などの組込みシンボルについては、大文字／小文字は無効です。

A、X、Y は予約語ですので、ラベルとしては使用できません。

ラベル

メモリ・ロケーションに使用されるシンボルは、ラベルといいます。

ロケーション・カウンタ

ロケーション・カウンタは、* といいます。例えば、次のとおりです。

```
          BRA       *       ; ループの継続
```

整数定数

IAR Systems のアセンブラはすべて３２ビットの２の補数内部演算を使用するため、整数には-2147483648～2147483647 の（符号付き）範囲があります。

定数は、負の数字を表すために任意指定の－（マイナス）符号を先頭につけて一連の数値として記述されます。

コンマや小数点は使用できません。

次のナンバー・ベースがサポートされています。

１６進数

１６進数は、次のいずれかの形式で記述できます。

形式	例	値
16 進数 H	**20H**	10 進数で 32 *
H'16 進数	**H'1A**	10 進数で 26
0x16 進数	**0x2A**	10 進数で 42
$16 進数	$80	10 進数で 128

＊　最初の桁がＡ～Ｆの場合は、先頭に０をいれてください。例えば、0AH のようにします。

８進数

８進数は、次のように記述します。

形式	例	値
８進数 Q	**10Q**	10 進数で８

１０進数

１０進数は、次のいずれかの形式で記述できます。

形式	例	値
数値	**123**	10 進数で 123
D'数値	**D'42**	10 進数で 42

２進数

２進数は、次のいずれかの形式で記述できます。

形式	例	値
２進数 B	**10B**	10 進数で２
B'２進数	**B'101**	10 進数で５

ASCII 文字定数

ASCII 文字定数は、引用符で囲まれた０～４文字間で構成されます。ASCII 文字列では、印刷可能な文字とスペースだけを使用することができます。

引用文字自体をアクセスする場合は、２つの連続した引用符を使用してください。

形式	値
'ABCD'	ABCD（４文字）
'A''B'	A'B
''（引用符２つ）	空ストリング（値＝０）

実数定数

740 アセンブラは実数を定数として受け付け、それを IEEE 単精度（符号付き３２ビット）実数形式に変換します。

浮動小数点数は、次の形式で記述されます。

"[+|-][数値].[数値][{E|e}[+|-}数値]

次に、有効な例をいくつか示します。

形式	値
“10.23	1.023×10^{1}
“1.23456E-24	1.23456×10^{-24}
“1.0E3	1.0×10^{3}

実数定数では、空白やタブは使用できません。

式の中で使用された場合、浮動小数点数は意味のある結果を示さないことに注意してください。

定義済みシンボル

740 アセンブラは、アセンブラ・ソース・ファイルでの使用のために一連のシンボルを

定義します。シンボルはカレント・アセンブリに関する情報を提供します。これによって、ユーザはプリプロセッサ疑似命令でシンボルを調べたり、アセンブルされたコードにシンボルを組み込んだりすることができます。

シンボル	値
__DATE__	**Mmm dd yyy** 形式で表される現在の日付
__FILE__	カレント・ソース・ファイル名
__IAR_SYSTEMS_ASM	IAR アセンブリ識別子
__LINE__	カレント・ソース行番号
__TID__	２バイトから成るターゲット識別。上位バイトはターゲット識別であり、740 の場合は 0x19 である。下位バイトは、プロセッサ・オプション *16 である。したがって、値は下表のようになる。
__TIME__	**hh:mm:ss** 形式で表される現在の時間

プロセッサ・オプション	値
-v0	**0x1900**
-v1	**0x1910**
-v2	**0x1920**

コードへのシンボル値の組込み

コードにシンボル値をいれるには、データ定義疑似命令の１つでシンボルを使用します。

例えば、プログラムによってアセンブリの時間とデータを文字列として表示させるには、次のようにします。

```
tanddstring                     ; 時間と日付のラベル
        BYTE    __TIME__,",",__DATE__,0
        ...
        LDX     #tanddstring    ; 文字列アドレスのロード
        JSR     printstring_X   ; 文字列を出力するためのルーチン
```

条件付きアセンブリでのシンボルのテスト

アセンブリ時にシンボルを調べるには、条件付きアセンブリ疑似命令の１つを使用します。

例えば、740 ファミリ・メンバーの１つで使用するために記述されたソース・ファイルで、MUL 命令をもたないバージョンでソフトウェア乗算ルーチンを使用したいと思う場合があります。そのような場合は、次のように__TID__シンボルを使用します。

```
#if __TID__& $10
PHA                     ; 'multiply'ルーチンのスタック・スペースの予約

JSR multiply
BYTE   operand          ; 'multiply'ルーチン用の行内データ

#else
MUL operand
#endif
```

アドレス演算

740 プロセッサの 19 アドレス指定モードの要件は、単にソース形式の４種類のアドレスによって満たされます。すなわち、即値バイト・アドレス、非即値バイト・アドレス、ビット・アドレス、相対アドレスです。

即値バイト・アドレス

即時モードなどのアドレス指定モードを使用する命令では、即値バイト・アドレスを使用します。

即値バイト・アドレスを指定するには、-128～255 の範囲で値を与えます。アセンブラは、（あるいは値がリンク時まで分からない場合は、リンカが、）値がこの範囲外だと検知すると、エラーを発行します。

例：

```
LDA   #255    ; アセンブラにより受け付けられる。
LDA   #256    ; アセンブラがエラーを発行する。
```

非即値バイト・アドレス

絶対、ゼロ・ページ、特殊ページなどのアドレス指定モードを使用する命令では、非即値バイト・アドレスを使用します。

非即値バイト・アドレスを指定するには、０～65535 の範囲で値を与えます。アセンブラは、（あるいは値がリンク時まで分からない場合は、リンカが、）値がこの範囲外だと検知すると、エラーを発行します。

命令には８ビットと１６ビットのアドレス指定モードの両方があり、与えられた値がアセンブリ時に分かる場合、アセンブラは自動的にアドレス指定モードを選択します。すなわち、値が０～255 の範囲にある場合は８ビット・アドレス指定モードを、そうでない場合は１６ビット・アドレス指定モードを選択します。

例：

```
LDA   255   ; ８ビット・アドレス指定が選択される。
LDA   256   ; １６ビット・アドレス指定が選択される。
```

未知の値

命令には８ビットと１６ビットのアドレス指定モードの両方がありますが、与えられた値がアセンブリ時に分からない場合、アセンブラは省略値の８ビット・アドレス指定モードを選択します。これは最小かつ最速のコードを与えますが、値が判明した場合、それは０～255 の範囲にあることを要求します。この値が８ビットの範囲を外れている場合、リンカはエラーを発行します。

例えば、以下の例では、count が０～255 の範囲を外れている場合、リンカはエラーを発行します。

```
EXTERN    count
LDA       count
```

このエラー発生を回避するには、以下に説明するように NP: 属性を使用してこのアドレスが 16 ビットと見なされるように指定するか、２３ページの第３章「アセンブラ・オプションのリファレンス」に説明するように-uN コマンド行オプションを使用して全ての未知の値を１６ビットと見なすように指定することができます。

明示アドレス長さ

ほとんどの場合、アセンブラが自動的に８ビット、あるいは１６ビット・アドレス指定モードを選択することに不満はないと思います。しかしながら、一方 ZP: 属性（ゼロ・ページの場合、８ビット）や NP: 属性（非ゼロ・ページの場合、１６ビット）を使用してアドレス長さを指定することもできます。この場合、式の前にこれらの属性の１つを置きます。

例えば、アセンブリ時には分からない値をもつ式で（省略値の８ビット・モードの選択を無効化して）１６ビット・アドレス指定を指定するには、次のように NP: 属性を使用します。

```
EXTERN  count          ; アセンブリ時には未知の１６ビット値
LDA     NP:count
```

あるいは、８ビット・アドレス指定を指定してアドレス値が目的どおり８ビット・モードの範囲にあるかどうかを確認するには、次のように ZP: 属性を使用します。

```
LDA     ZP:temp        ; temp はゼロ・ページ位置でなければなりません。
```

アセンブラはまた、完全を記するために１６ビット・アドレス指定のみをもつ命令では NP: 属性を、また８ビット・アドレス指定のみをもつ命令では ZP: 属性を使用できるようにもしています。例えば、次のとおりです。

```
TST ZP:label          ; label が８ビット範囲でなければならないことを明確にする。
JMP NP:label          ; label が 16 ビット範囲でなければならな
```

いことを明確にする。

特殊ページ・アドレス

特殊ページ・アドレスを指定するには、特殊ページにある非即値バイト・アドレス値（特殊ページ・シグニファイア＼が先頭にある）を与えます。特殊ページのロケーションは$1FXX, $3FXX、あるいは$FFXX で、プロセッサ・バージョンによって異なります。

アセンブラは、値の下位バイト（$XX）をもつ特殊ページ命令をオペランドとして生成

します。アセンブラは値が特殊ページ内にあるかどうかをチェックしませんので、自分でチェックするように注意してください。

ビット・アドレス

アキュミュレータ・ビット（３ビット範囲をもつ）やゼロ・ページ・ビット（１１ビット範囲をもつ）などのアドレス指定モードを使用する命令では、ビット・アドレスを使用します。

３ビットのビット・アドレスを指定するには、０～７の範囲でバイト内のビットの番号を与えます。

例：

```
CLB 7,A               ; 受け付けられる
CLB 8,A               ; '8'は範囲外のため、エラーが発生する。
```

１１ビットのビット・アドレスを指定するには、０～2047 の範囲でバイト０のビット０からカウントしたビットの番号を、あるいは０～255 の範囲でバイト内のビットの番号およびその後に続くバイトの番号の２つの値を与えます。

例：

```
BBC     81，label           ；ビット 81 を調べる。
BBC     1，10，label        ；バイト 10 のビット１を調べる。
```

これらの２つの命令は２種類の方法で指定されていますが、同じビットをアドレス指定します。

前述したように、アセンブラは、（あるいは値がリンク時まで分からない場合は、リンカが、）値がこの範囲外だと検知すると、エラーを発行します。

相対アドレス

分岐命令では相対アドレスを使用します。

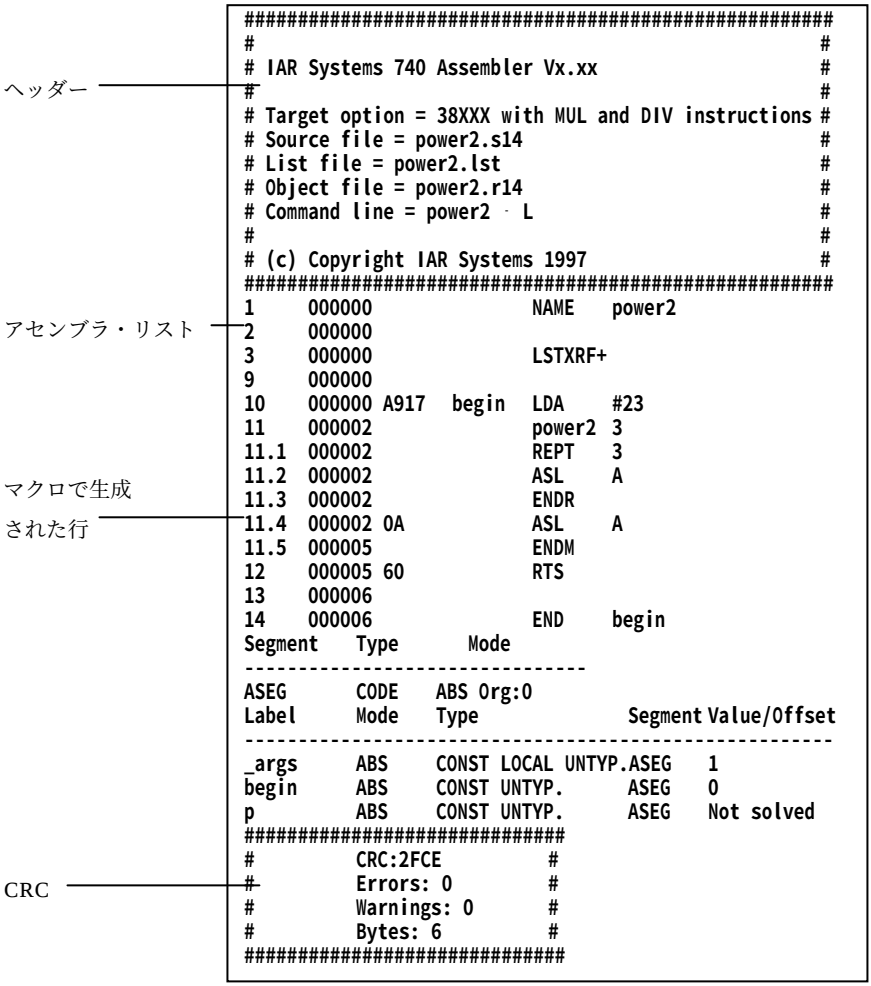
相対アドレスを指定するには、*-128 および sizeof（命令）～*+127 および sizeof（命令）の範囲で値を与えます。ここで、* はプログラム・カウンタの値を示し、sizeof（命令）は BBC ゼロ・ページおよび BBS ゼロ・ページ命令の場合で３、その他の分岐の場合はすべて２です。通常、これはラベルの形式をとります。アセンブラは、（あるいは値がリンク時まで分からない場合は、リンカが、）値がこの範囲外だと検知すると、エラーを発行します。

例：

```
BBC     label               ；label が 128 バイト以内であれば、受け付けられる。
...
label
```

リスト形式

740 アセンブラ・リストの形式は、次のとおりです。




```
#########################################################
#                                                       #
# IAR Systems 740 Assembler Vx.xx                       #
#                                                       #
# Target option = 38XXX with MUL and DIV instructions #
# Source file = power2.s14                              #
# List file = power2.lst                                #
# Object file = power2.r14                              #
# Command line = power2 -L                              #
#                                                       #
# (c) Copyright IAR Systems 1997                        #
#########################################################
1     000000                NAME    power2
2     000000
3     000000                LSTXRF+
9     000000
10    000000 A917   begin   LDA     #23
11    000002                power2  3
11.1  000002                REPT    3
11.2  000002                ASL     A
11.3  000002                ENDR
11.4  000002 0A             ASL     A
11.5  000005                ENDM
12    000005 60             RTS
13    000006
14    000006                END     begin
Segment    Type       Mode
-------------------------------
ASEG       CODE    ABS Org:0
Label      Mode    Type              Segment Value/Offset
-----------------------------------------------------------
_args      ABS     CONST LOCAL UNTYP.ASEG    1
begin      ABS     CONST UNTYP.      ASEG    0
p          ABS     CONST UNTYP.      ASEG    Not solved
###############################
#          CRC:2FCE           #
#          Errors: 0          #
#          Warnings: 0        #
#          Bytes: 6           #
###############################
```

アセンブリ・パラメータをもつヘッダーは、端末以外のファイルに向けて送信されるリストだけに出力されます。

アセンブリ・リスト情報は、４つのフィールドに置かれます。

```
10    000000 A917   begin   LDA     #23
11    000002                power2  3
11.1  000002                REPT    3
11.2  000002                ASL     A
11.3  000002                ENDR
11.4  000002 0A             ASL     A
```

アドレス　　ソース行

ソース行番号　　データ

ソース行番号

ソース・ファイルの行番号

マクロで生成された行は、リストされた場合にソース行番号フィールドに・[ドット]をもちます。

アドレス・フィールド、データ・フィールド

これらは、１６進表示法では常にリストされます。

ソース行

ソース・ファイル行をリストします。

シンボルと相互参照テーブル

LSTXRF+疑似命令が組み込まれていたり、Xコマンド行オプションが指定されている場合は、次のシンボルと相互参照テーブルが生成されます。

セグメント

シンボル

```
Segment    Type       Mode
-------------------------------
ASEG       CODE       ABS Org:0
Label      Mode       Type              Segment
           Value/Offset
-----------------------------------------------------------------
_args      ABS        CONST LOCAL UNTYP ASEG          1
begin      ABS        CONST UNTYP.      ASEG          0
p          ABS        CONST UNTYP.      ASEG          Not solved
```

相互参照テーブル内の各シンボルには、次の項目が提供されます。

項目	内容
ラベル	ラベルのユーザ定義名
モード	ABS（絶対モード）、REL（相対モード）
型	ラベルの型
セグメント	このラベルが相対的に定義されるセグメント名
値／オフセット	カレント・セグメントの始まりに関連するカレント・モジュール内のラベルの値（アドレス）

出力形式

再配置可能な絶対出力は、全てのアセンブラについて同じ形式になります。これは、オブジェクト・コードが常に IAR Systems XLINK リンカによって処理されるようになっているからです。

しかしながら、XLINK からの出力は、通常独立系ソースからのスタンドアロン形エミュレータや PROM プログラマとともにチップ・ベンダーのデバッガ・プログラム（モニター）と互換性のある絶対形式になります。

第５章　アセンブラ演算子の概要

この章は、優先順位に従って分類されたアセンブラ演算子についてまとめています。次の章には、演算子をアルファベット順に並べた参照リストを示します。

演算子の優先順序

各演算子には、優先順位番号が割り付けられています。この番号は、演算子やそのオペランドが評価される順番を決定します。優先順位の番号は、１（最高の優先順位、すなわち最も早く評価される）～７（最低の優先順位、すなわち最後に評価される）の範囲にあります。

式の評価方法は、次の規則に従って決められます。

- 最高の優先順位（最も小さい番号）の演算子が最初に評価され、その後次に高い優先順位をもつ演算子が評価されるという具合いに、最低の優先順位の演算子が評価されるまで続けられます。
- 同等の優先順位をもつ演算子は、式の中で左から右へ評価されます。
- 演算子やオペランドをグループに分けたり、式を評価する順番を制御するには、カッコ（、）を使用します。例えば、次式は１と求められます。

```
7/(1+(2*3))
```

以降に、演算子を優先順位に従ってまとめています。同義語が適用される場合には、演算子名の後にカッコで示しています。

単項演算子－１

-	単項マイナス
+	単項プラス
NOT	論理 NOT
LOW	下位バイト
HIGH	上位バイト
DATE	現在の日付／時間
SFB	セグメントの開始
SFE	セグメントの終り
SIZEOF	セグメント・サイズ
BINNOT(～)	ビット NOT

乗算算術演算子－３

*	乗算
/	除算
MOD(%)	モジュロ
SHR (>>)	論理右シフト
SHL (<<)	論理左シフト

加算算術演算子－４

+	加算
-	減算

AND 演算子－５

AND (&&)	論理 AND
BINAND(&)	ビット AND

OR 演算子－６

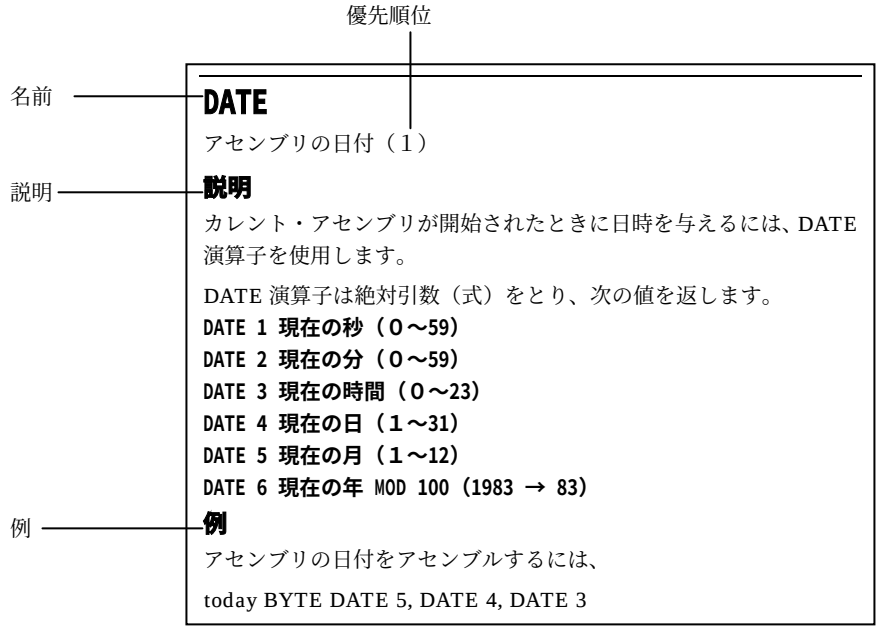
OR (||)	論理 OR
XOR	排他的論理 OR
BINOR (|)	ビット OR
BINXOR (^)	ビット XOR

比較演算子－７

EQ (=,==)	等価
NE (<>,!=)	不等価
GT (>)	右不等
LT (<)	左不等
UGT	符号無し右不等
ULT	符号無し左不等
GE (>=)	等価右不等
LE (<=)	等価左不等

第６章　アセンブラ演算子のリファレンス

この章ではアセンブラ演算子をアルファベット順にリストし、各演算子を詳細に説明します。



名前

演算子名、また適切な演算子の同義語がある場合はそれを示し、続いて演算子の優先順位を示します。

演算子名の後に、その説明を記載します。

説明

演算子の最も一般的な使用を含む詳細な説明です。

例

演算子の一般的なアプリケーションを示し、また特殊な事例を明確にした例です。

*

乗算（３）

説明

＊は、その２つのオペランドの積を生成します。オペランドは符号付き３２ビット整数としてとられ、結果も符号付き３２ビット整数となります。

例

```
2*2 → 4
-2*2 → -4
```

+

単項プラス（１）

説明

単項プラス演算子

＋

加算（４）

説明

＋加算演算子は、この演算子を囲む２つのオペランドの和を生成します。オペランドは符号付き３２ビット整数としてとられ、結果も符号付き３２ビット整数となります。

例

```
92+19 → 111
-2+2 → 0
-2+-2 → -4
```

ー

単項マイナス（１）

説明

単項マイナス演算子は、そのオペランドの算術否定を行います。

オペランドは符号付き３２ビット整数として解釈され、演算子の結果はその整数の２の補数否定となります。

－

減算（４）

説明

減算演算子は、右オペランドが左オペランドから差し引かれた差を生成します。オペランドは符号付き３２ビット整数としてとられ、結果も符号付き３２ビット整数となります。

例

92-19 → 73
-2-2 → -4
-2--2 → 0

/

除算（３）

説明

/ は、左オペランドが右オペランドによって割り算された商を生成します。オペランドは符号付き３２ビット整数としてとられ、結果も符号付き３２ビット整数となります。

例

8/2 → 4
-12/3 → 4

AND (&&)

論理 AND（５）

説明

２つの整数オペランド間で論理 AND を行うには、AND を使用します。両オペランドが０以外の数値の場合、結果は１となり、それ以外の場合は０となります。

例

1010B AND 0011B → 1
1010B AND 0101B → 1
1010B AND 0000B → 0

BINAND (&)

ビット AND（5）

説明

整数オペランド間でビット AND を行うには、BINAND を使用します。

例

```
1010B BINAND 0011B → 0010B
1010B BINAND 0101B → 0000B
1010B BINAND 0000B → 0000B
```

BINNOT (~)

ビット NOT（1）

説明

オペランドにビット NOT を実行するには、BINNOT を使用します。

例

```
BINNOT 1010B → 11111111111111111111111111110101B
```

BINOR (|)

ビット OR（6）

説明

オペランドにビット OR を実行するには、BINOR を使用します。

例

```
1010B BINOR 0101B → 1111B
1010B BINOR 0000B → 1010B
```

BINXOR (^)

排他的ビット OR（6）

説明

オペランドにビット XOR を実行するには、BINXOR を使用します。

例

```
1010B BINXOR 0101B → 1111B
1010B BINXOR 0011B → 1001B
```

DATE

アセンブリの日付（1）

説明

カレント・アセンブリが開始されたときに日時を与えるには、DATE 演算子を使用します。

DATE 演算子は絶対引数（式）をとり、次の値を返します。

DATE 1 現在の秒（0～59）

DATE 2 現在の分（0～59）

DATE 3 現在の時間（0～23）

DATE 4 現在の日（1～31）

DATE 5 現在の月（1～12）

DATE 6 現在の年 MOD 100（1983 → 83）

例

アセンブリの日付をアセンブルするには、

```
today BYTE DATE 5, DATE 4, DATE 3
```

EQ (=,==)

等価（7）

説明

EQ は、その2つのオペランドが同じ値であれば1（真）、同じ値でないならば0（偽）と評価します。

例

```
1 EQ 2 → 0
2 EQ 2 → 1
'ABC' EQ 'ABCD' → 0
```

GE (>=)

等価右不等（7）

説明

GE は、左オペランドが右オペランドと等しい、あるいはそれよりも大きな数値をもつならば1（真）と評価します。

例

```
1 GE 2 → 0
2 GE 1 → 1
1 GE 1 → 1
```

GT (>)

右不等（7）

説明

GT は、左オペランドが右オペランドよりも大きな数値をもつならば1（真）と評価します。

例

```
-1 GT 1 → 0
2 GT 1 → 1
1 GT 1 → 0
```

HIGH

第2バイト（1）

説明

HIGH は、符号なし１６ビット整数値として解釈される単一オペランドをその右側に移します。結果は、このオペランドの上位バイトの符号なし８ビット整数値となります。

例

```
HIGH 1234ABCDh → ABh
```

LE (<=)

等価左不等（7）

説明

LE は、左オペランドが右オペランドより小さい、あるいは等しい数値をもつならば1（真）と評価します。

例

```
1 LE 2 → 1
2 LE 1 → 0
1 LE 1 → 1
```

LOW

下位バイト（1）

説明

LOW は、符号なし３２ビット整数値として解釈される単一オペランドをとります。結果は、このオペランドの下位バイトの符号なし８ビット整数値となります。

例

```
LOW 1234ABCDh → CDh
```

LT (<)

左不等（7）

説明

LT は、左オペランドが右オペランドよりも小さい数値をもつならば1（真）と評価します。

例

```
-1 LT 2 → 1
2 LT 1 → 0
2 LT 2 → 0
```

MOD (%)

モジュロ（3）

説明

MOD は、右オペランドによる左オペランドの整数除算からの剰余を生成します。オペランドは符号付き３２ビット整数としてとられ、結果も符号付き３２ビット整数となります。

X MOD Y は、整数除算を使用する X-Y*(X/Y)に相当します。

例

```
2 MOD 2 → 0
12 MOD 7 → 5
3 MOD 2 → 1
```

NE (<>,!=)

不等価（7）

説明

NE は、その２つのオペランドが同じ値であれば０（偽）、同じ値でないならば１（真）と評価します。

例

```
1 NE 2 → 1
2 NE 2 → 0
'A' NE 'B' → 1
```

NOT (!)

論理 NOT（1）

説明

論理引数を否定するには、NOT を使用します。

例

```
NOT 0101B → 0
NOT 0000B → 1
```

OR (||)

論理 OR（6）

説明

２つの整数オペランド間で論理 OR を行うには、OR を使用します。

例

```
1010B OR 0000B → 1
0000B OR 0000B → 0
```

SFB

セグメントの開始（1）

構文

SFB(セグメント[{+ | -} オフセット])

パラメータ

セグメント	再配置可能なセグメントの名前
オフセット	任意指定可能な開始アドレスからのオフセット。オフセットを省略した場合、カッコは任意指定です。

説明

SFB は、単一オペランドをその右側で受け付けます。オペランドは、再配置可能なセグントの名前でなければなりません。演算子は、そのセグメントの最初のバイトの絶対アドレスと評価します。この評価は、リンク時に行われます。

例

```
        NAME    demo
        RSEG    CODE
start   VAR     SFB(CODE)
```

上記のコードが他の多数のモジュールとリンクされている場合でも、start はセグメントの最初のバイトのアドレスに設定されます。

SFE

セグメントの終り（1）

構文

SFE(セグメント[{+ | -} オフセット])

パラメータ

セグメント　　再配置可能なセグメントの名前

オフセット　　任意指定可能な開始アドレスからのオフセット。オフセットを省略した場合、カッコは任意指定です。

説明

SFE は、単一オペランドをその右側で受け付けます。オペランドは、再配置可能なセグメントの名前でなければなりません。演算子は、セグメント開始アドレスおよびセグメント・サイズと評価します。この評価は、リンク時に行われます。

例

```
        NAME    demo
        RSEG    CODE
end     VAR     SFE(CODE)
```

上記のコードが他の多数のモジュールとリンクされている場合でも、end はセグメントの最初のバイトのアドレスに設定されます。

SHL (<<)

論理左シフト（3）

説明

左オペランドを左にシフトするには、SHL を使用します。シフトさせるビット数は、右オペランドによって指定され、0～32 の間の整数値として解釈されます。

例：

```
00011100B SHL 3 → 11100000B
00000111111111111B SHL 5 → 1111111111100000B
14 SHL 1 → 28
```

SHR (>>)

論理右シフト（3）

説明

左オペランドを右にシフトするには、SHR を使用します。シフトさせるビット数は、
右オペランドによって指定され、0～32 の間の整数値として解釈されます。

例：

```
01110000B SHR 3 → 00001110B
1111111111111111B SHR 20 → 0
14 SHR 1 → 7
```

SIZEOF

セグメント・サイズ（1）

構文

SIZEOF セグメント

パラメータ

セグメント　　再配置可能なセグメントの名前

説明

SIZEOF は、その引数の SFE-SFB を生成します。引数は、再配置可能なセグメントの名前でなければなりません。つまり、セグメントのバイト数が計算されます。これは、モジュールがリンクされたときに行われます。

例

```
        NAME    demo
        RSEG    CODE
size    .SET    SIZEOF CODE
```

size をセグメント code のサイズに設定します。

UGT

符号なし右不等（7）

説明

UGT は、左オペランドが右オペランドよりも大きな絶対値をもつならば1（真）と評価します。

例

```
2 UGT 1 → 1
-1 UGT 1 → 0
```

ULT

符号なし左不等（7）

説明

ULT は、左オペランドが右オペランドよりも小さい絶対値をもつならば1（真）と評価します。

例

```
1 ULT 2 → 1
-1 ULT 2 → 0
```

XOR

排他的論理 OR（6）

説明

2つのオペランドで排他的 XOR を行うには、XOR を使用します。

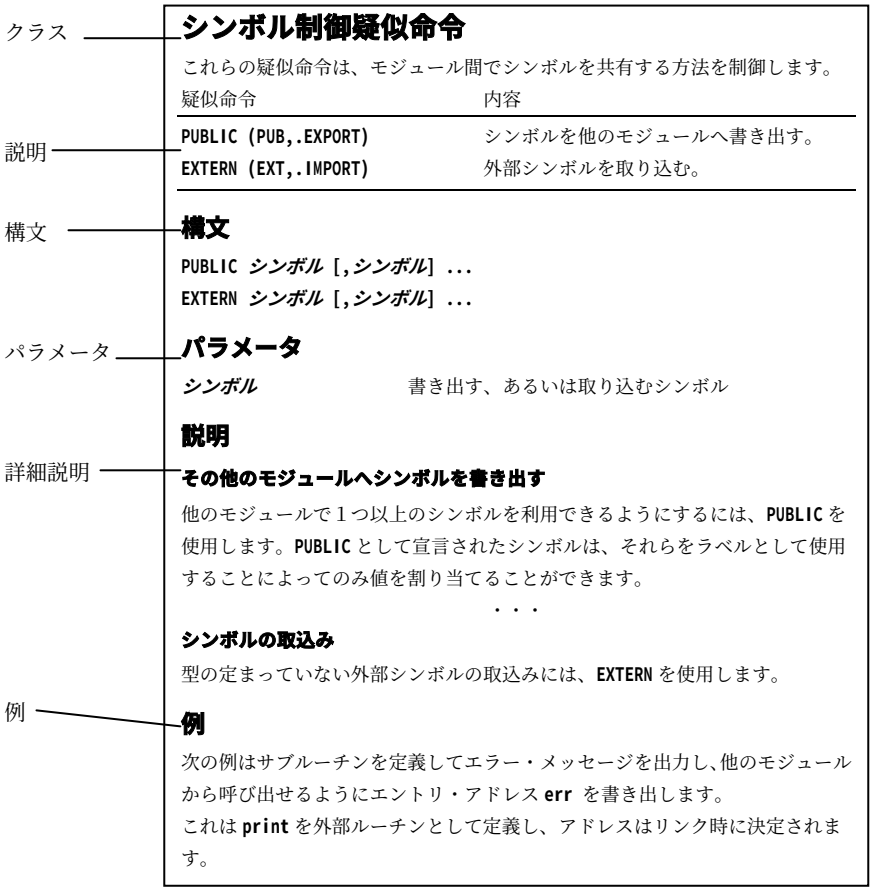
例

```
0101B XOR 1010B → 0
0101B XOR 0000B → 1
```

第７章　アセンブリ疑似命令のリファレンス

この章は、機能に従って分類され、演算の詳細説明および各疑似命令で使用できるオプションを説明した 740 疑似命令をリストします。

各項の形式は、次のようになっています。



クラス

疑似命令のクラスです。

クラスの後にはクラスの説明と、そのクラスの各疑似命令の説明が続きます。

構文

各疑似命令の詳細な構文定義です。

パラメータ

構文定義の各パラメータの詳細です。

説明

各疑似命令の最も一般的な使用法を含む詳細説明です。ここには、疑似命令の有用性に関する情報、特殊な条件、一般的な注意点なども記述しています。

例

疑似命令の一般的なアプリケーションを示し、また特殊な事例を明確にした例です。

構文上の表記規則

構文の定義には、次の規則を使用します。

入力項目を表したパラメータは、イタリック体で表記します。したがって、

ORG *式*

の例では、式は任意の式を表しています。

任意指定のパラメータは、大カッコの中に示します。したがって、

END [*式*]

の例では、式パラメータは任意指定となります。

省略符号は、その前の項目が任意の回数だけ繰り返されることを表しています。例えば、

LOCAL *シンボル* [,*シンボル*] ...

は、**LOCAL** の後に１つ以上のシンボルをコンマで区切って続けることができることを表しています。

選択肢は、垂直線で区切って中カッコ {} で囲っています。例えば、

LSTOUT{+ | -}

は、疑似命令の後に＋か－のいずれかが続かなければならないことを示しています。

ラベルとコメント

疑似命令の先頭にラベルをつけなければならない場合は、次のように構文で表示されます。

ラベル* .SET *式

その他の疑似命令はすべて、カレント・ロケーション・カウンタ（PLC）の値と型をもつ任意指定のラベルを先頭につけることができます。これは、明確さを記すために各構文の定義には含まれていません。

さらに、特に明示的に指定していない限り、全ての疑似命令には；（セミコロン）を先頭につけてコメントを続けることができます。

パラメータ

下表に、最も一般的に使用されるパラメータの型の適切な形式を示します。

パラメータ	構成
シンボル	アセンブラ・シンボル
ラベル	記号ラベル
式	式。３６ページの「式と演算子」を参照のこと。

疑似命令の名前

全ての疑似命令は、メーカーのアセンブリとの互換性をとるために疑似命令の前に .（ピリオド）を置くことができます。

モジュール制御疑似命令

ソース・プログラム・モジュールの開始や終りを記したり、それらに名前や型を割り付けるには、モジュール制御疑似命令を使用します。

疑似命令	内容
NAME (PROGRAM)	プログラム・モジュールの開始
MODULE (LIBRARY)	ライブラリ・モジュールの開始
ENDMOD (.ENDMOD)	カレント・モジュールのアセンブリの終了
END (.END)	ファイルの中の最後のモジュールのアセンブリの終了

構文

NAME *シンボル* [(*式*)]

MODULE *シンボル* [(*式*)]

ENDMOD [*ラベル*]

END [*ラベル*]

パラメータ

シンボル XLIB がモジュールを参照する場合に使用するモジュールに割り付けられた名前

式 IAR Cコンパイラが使用する任意指定の式（0～255）

ラベル アセンブリ時に決められる式やラベル。これは、プログラム入力アドレスとしてオブジェクト・コードで出力されます。

説明

プログラム・モジュールの開始

プログラム・モジュールを開始し、XLINK や XLIB が以降で参照することができるように名前を割り付けるには NAME を使用します。

プログラム・モジュールは、その他のモジュールで参照されない場合でも XLINK によって無条件にリンクされます。

ライブラリ・モジュールの開始

各モジュールがまた単一ルーチンを表すことがある（高レベル言語用のランタイム・システムのような）多数の小さなモジュールをもったライブラリを生成するには、**MODULE** を使用します。マルチモジュール機能によって、必要なソース・ファイルやオブジェクト・ファイルの数を大幅に削減することができます。

モジュールの公用シンボルがその他のモジュールで参照される場合、ライブラリ・モジュールはリンクされたコードのみにコピーされます。

モジュールの終了

モジュールの終了を定義するには、**ENDMOD** を使用します。

最後のモジュールの終了

ソース・ファイルの終りを指示するには、**END** を使用します。**END** 疑似命令の後の行は、無視されます。

プログラム・エントリは再配置可能なものか、あるいは絶対的なもの（外部的なものは使用できません）かのどちらかでなければならず、一部の１６進絶対出力形式とともに XLINK ロード・マップにも現れます。

マルチモジュール・アセンブリには、次の規則が適用されます。

- 新しいモジュールの始まりでは、**DEFINE**、**#define**、あるいは **MACRO** によって生成されたものを除く全てのユーザ・シンボルは削除され、ロケーション・カウンタはクリアされ、絶対モードが選択されます。
- リスト制御疑似命令は、アセンブリを通して効果があります。

END は常に最後のモジュールで使用され、**ENDMOD** と **MODULE** 疑似命令間にソース行がなければなりません（ただし、コメントやリスト制御疑似命令は除きます）。

NAME や **MODULE** 疑似命令が抜けている場合は、モジュールにはソース・ファイルの名前とプログラム属性が割り付けられます。

例

次の例は、３つのモジュールを定義します。

```
MODULE
.
. Module #1
.
ENDMOD
MODULE
.
. Module #2
.
ENDMOD
MODULE
.
. Last module
END
```

シンボル制御疑似命令

これらの疑似命令は、モジュール間でシンボルを共有する方法を制御します。

疑似命令	内容
PUBLIC (PUB,.EXPORT)	シンボルを他のモジュールへ書き出す。
EXTERN (EXT,.IMPORT)	外部シンボルを取り込む。

構文

PUBLIC ***シンボル*** **[,*シンボル*] ...**

EXTERN ***シンボル*** **[,*シンボル*] ...**

パラメータ

シンボル 書き出す、あるいは取り込むシンボル

説明

その他のモジュールへシンボルを書き出す

他のモジュールで１つ以上のシンボルを利用できるようにするには、**PUBLIC** を使用します。**PUBLIC** として宣言されたシンボルは、それらをラベルとして使用することによってのみ値を割り当てることができます。**PUBLIC** 宣言されたシンボルは再配置可能なものか、あるいは絶対的なものであり、式の中でも（その他のシンボルの場合と同じ規則で）使用することができます。

PUBLIC 疑似命令は、常に完全な３２ビット値を書き出します。これによって、８ビット・プロセッサや１６ビット・プロセッサ用のアセンブラにもグローバル３２ビット定数の使用が可能になります。**LOW**、**HIGH**、**>>**、**<<** 演算子によって、このような定数のどの部分もが８ビット、あるいは１６ビットのレジスタやワードにロードすることができます。モジュール内の **PUBLIC** 宣言シンボルの数については、なんら制限はありません。

シンボルの取込み

型の定まっていない外部シンボルの取込みには、**EXTERN** を使用します。

例

次の例はサブルーチンを定義してエラー・メッセージを出力し、他のモジュールから呼び出せるようにエントリ・アドレス **err** を書き出します。

これは **print** を外部ルーチンとして定義し、アドレスはリンク時に決定されます。

```
    1  000000              NAME     error
    2  000000              EXTERN   print
    3  000000              PUBLIC   err
    4  000000
    5  000000 20....  err  JSR      print
    6  000003 2A2A2A20     BYTE     "*** Error ***"
    7  000011 60           RTS
    8  000012
    9  000012              END      err
```

（原文どおり）

セグメント制御疑似命令

セグメント疑似命令は、コードやデータの生成方法を制御します。

疑似命令	内容
ASEG	絶対セグメントを開始する。
RSEG (SECTION)	再配置可能なセグメントを開始する。
STACK	スタック・セグメントを開始する。
COMMON	共通セグメントを開始する。
ORG	ロケーション・カウンタを設定する。
ALIGNRAM	単にアップデートすることによってプログラム・カウンタを調整する。バイトは挿入されない。
ALIGN	０を入れたバイトを挿入することによってプログラム・カウンタを合わせる。
EVEN	必要に応じて０を入れたバイトを挿入することによってプログラム・カウンタを偶数アドレスに合わせる。
ODD	必要に応じて０をいれたバイトを挿入することによってプログラム・カウンタを奇数アドレスに合わせる。

構文

ASEG [*スタート* [(*整合*)]]

RSEG *セグメント* [:*型*] [(*整合*)]

STACK *セグメント* [:*型*] [(*整合*)]

COMMON *セグメント* [:*型*] [(*整合*)]

ORG *式*

ALIGN [*整合*]

ALIGNRAM [*整合*]

EVEN

ODD

パラメータ

スタート	絶対セグメントの始まりで **ORG** 疑似命令を使用するのと同じ効果をもつ開始アドレス
セグメント	セグメント名
型	メモリ・タイプ **UNTYPED**（省略値）、**CODE**, **DATA**, **NPAGE**, **ZPAGE** のうちのいずれか１つ さらに、IAR Cコンパイラとの互換性を得るために次のタイプを用意しています。 **XDATA**, **IDATA**, **BIT**, **REGISTER**, **CONST**
式	ロケーション・カウンタを設定するアドレス
整合	０～31 の範囲でアドレスが合わせられなければならない２の累乗

説明

絶対セグメントの開始

モジュールの始まりでデフォルトとなるアセンブリの絶対モードを設定するには、**ASEG** を使用します。

パラメータを省略すると、最初のセグメントの開始アドレスは０となり、後続のセグメントは前のセグメントの最後のアドレスの後に続きます。

再配置可能なセグメントの開始

アセンブリのカレント・モードを再配置可能なアセンブリ・モードに設定するには、**RSEG** を使用します。アセンブリは、全てのセグメントについて（ゼロに初期設定された）個別のロケーション・カウンタを維持します。これによって、現在のセグメント・ロケーション・カウンタを保存しなくてもいつでもセグメントやモードを切り換えることができます。

最大 256 個の一意の再配置可能なセグメントを１モジュールに定義することができます。

スタック・セグメントの開始

上位アドレスから下位アドレスまで割り当てられたコードやデータを割り付けるには、**STACK** を使用します（下位アドレスから上位アドレスへの割当を行う RSEG 疑似命令に対して）。

セグメントのコンテンツは、逆の順序で生成されることはありません。

共通セグメントの開始

同じ名前をもつ他のモジュールからの COMMON セグメントと同じ位置のメモリにデータを置くには、COMMON を使用します。すなわち、同じ名前の COMMON セグメントはすべて、メモリの同じ位置からスタートし、互いにオーバレイします。

明らかに、COMMON セグメント型はオーバレイされた実行可能型コードに使用すべきではありません。一般的には、多数の異なったルーチンがデータ用の再使用可能な共通領域を共有するようにしたい場合に適用されます。

COMMON セグメントには割込みベクトル・テーブルをもつのが実用的です。これによって、いくつかのルーチンからアクセスが可能になります。

COMMON セグメントの最終サイズは、このセグメントの最大サイズによって決まります。メモリの位置は、XLINK の -Z コマンドによって決まります。「IAR リンカ&ライブラリアン・ガイド」の「XLINK コマンドのリファレンス」を参照してください。

上記のいずれの疑似命令においても整合パラメータの指定は、同じ値をもつ ALIGN 疑似命令を含むことに相当します。

ロケーション・カウンタの設定

現在のセグメントのロケーション・カウンタを式の値に設定するには、ORG を使用します。任意指定のラベルは、新しいロケーション・カウンタの値や型を想定します。

式の結果は、現在のセグメントと同じ型のものでなければなりません。すなわち、式は絶対的なものですから、RSEG 中に ORG 10 を使用するのは有効ではありません。代わりに、ORG *+10 を使用してください。式には、順方向参照や外部参照があってはいけません。

全てのロケーション・カウンタは、アセンブリ・モジュールの始まりでゼロに設定されます。

セグメントの整合

プログラム・カウンタを指定アドレス境界に合わせるには、ALIGN を使用します。式は、プログラム・カウンタが合わせられなければならない２の累乗を与えます。

プログラム・カウンタを指定アドレス境界に合わせるには、ALIGNRAM を使用します。式は、プログラム・カウンタが合わせられなければならない２の累乗を与えます。

ALIGH については、プログラム・カウンタはインクレメントされるだけですから、ゼロは挿入されないことに注意してください。

プログラム・カウンタを偶数アドレスに合わせるには、**EVEN** を使用します。必要ならば0バイトを挿入します。これは、次のものに相当します。

```
ALIGN 1
```

プログラム・カウンタを奇数アドレスに合わせるには、**ODD** を使用します。必要ならば0バイトを挿入します。

例

絶対セグメントの開始

次の例では、ベクトルは$FFCA アドレスから始まる絶対セグメントを使用して、適切な 3760 割込みアドレスでベクトル・アドレスをアセンブルします。

```
        EXTERN    brkhandler,INT2handler,CNTR0handler
        EXTERN    Timer3handler,Timer2handler,Timer1handler

        ASEG
        ORG       $FFCA
        WORD      brkhandler
        WORD      CNTR0handler
        WORD      Timer3handler
        WORD      Timer2handler
        WORD      Timer1handler
```

再配置可能なセグメントの開始

以下の例では、最初の RSEG 疑似命令に続くデータは **table** という再配置可能なセグメントにおかれます。テーブルに6バイトのギャップを生成するには、**ORG** 疑似命令が使用されます。

2番目の **RSEG** 疑似命令に続くコードは、code という再配置可能なセグメントにおかれます。

```
        EXTERN    divrtn,mulrtn

        RSEG      table
        WORD      divrtn,mulrtn

        ORG       *+6
        BYTE      subrtn
```

```
          RSEG      code
subrtn    LDA       #1
          RTS

          END
```

スタック・セグメントの開始

次の例は、**rpnstack** という再配置可能なセグメントに 100 バイト・スタックを２つ定義します。

```
          STACK     rpnstack
parms     BLKB      100
opers     BLKB      100

          END
```

データは、上位アドレスから下位アドレスへ割り当てられます。

共通セグメントの開始

次の例は、変数をもった２つの共通セグメントを定義します。

```
          NAME      common1
          COMMON    data
count     BLKB      4
          ENDMOD

          NAME      common2
          COMMON    data
up        BLKB      1
          ORG       *+2
down      BLKB      1
          END
```

共通セグメントは同じ名前 data をもっているため、変数 **up**, **down** はメモリの同じ位置を４バイト変数 **count** の最初と最後のバイトとして参照します。

セグメントの整合

次の例は、ページ境界から始まる相対セグメントを定義します。

```
          NAME      aligntest
          EXTERN    entry1,entry2,entry3

subr      RSEG      table (8)
          ; Table must start on a page boundary
          WORD      entry1,entry2,entry3
          END       subr
```

次の例は、プログラム・カウンタを偶数アドレスに合わせます。

```
          NAME      even

          EVEN
subr      LSR       A
          RTS

          END       subr
```

値割当て疑似命令

値をシンボルに割り当てるには、これらの疑似命令を使用します。

疑似命令	内容
VAR (ASSIGN,.SET)	一時値を割り当てる。
EQU (ALIAS,=)	モジュールにローカルなパーマネント値を割り当てる。
DEFINE	ファイル全体に及ぶ値を定義する。
SFR	SFR ラベルを生成する。
SFRTYPE	SFR 属性を指定する。
LIMIT	値が指定範囲内にあるかどうかをチェックする。

構文

ラベル* VAR *式

ラベル* EQU *式

ラベル* = *式

ラベル* DEFINE *式

SFR *レジスタ* = *値*

SFRTYPE *レジスタ 属性* [,*属性*] = *値*

LIMIT *ラベル*, *最小*, *最大*, *メッセージ*

パラメータ

ラベル 定義するシンボル

式 シンボルに割り当てられた値

レジスタ 特殊機能レジスタ

属性 次のうちの１つ以上

- **READ** この SFR から読み取られる。
- **WRITE** この SFR に書き込まれる。
- **BYTE** SFR は、バイトとしてアクセスしなければならない。

値 SFR 値

最小*、*最大 ラベルで許される最小値、最大値

メッセージ シンボルが範囲を外れている場合に出力されるテキスト・メッセージ

説明

一時値の定義

マクロ変数との使用など再定義可能なシンボルを定義するには、**VAR** を使用します。**VAR** で定義されたシンボルは、**PUBLIC** 宣言することができません。

パーマネント・ローカル値の定義

シンボルに値を割り当てるには、**EQU** や**=** を使用します。

数字やオフセットを示すローカル・シンボルを生成するには、**EQU** を使用します

シンボルはそれが定義されたモジュールでのみ有効ですが、**PUBLIC** 疑似命令を使用して他のモジュールでも利用することができます。

他のモジュールからシンボルを取り込むには、**EXTERN** を使用します。

パーマネント・グローバル値の定義

ソース・ファイル内の全モジュールに通知しなければならないシンボルを定義するには **DEFINE** を使用します。

DEFINE で値を指定されたシンボルは、**PUBLIC** 疑似命令でファイル内のモジュールで利用可能にすることができます。

DEFINE で定義されたシンボルは、再定義することはできません。

特殊機能レジスタの定義

全ての属性をオンにして特殊機能レジスタ・ラベルを生成するには、**SFR** を、指定された属性をもつ特殊機能レジスタ・ラベルを生成するには、**SFRTYPE** を使用します。

シンボル値のチェック

シンボルが指定範囲内にあるかどうかをチェックするには、**LIMIT** を使用します。シンボルに指定範囲外の値が割り当てられている場合は、エラー・メッセージが出力されます。

最小、最大 ラベルに許されている最小値、最大値

メッセージ シンボルが範囲外の場合に出力されるテキスト・メッセージ

チェックは値が決定されるとすぐに実行されますが、式に外部参照がある場合はリンク時に行われます。最小、最大の式は、順方向ラベルや外部ラベルの参照を含むことはできません。すなわち、これらの式は現れた時点で決定されなければなりません。

例

シンボルの再定義

次の例は **REPT** ループのシンボル **const** を再定義するために **VAR** を使用して、３の最初の８乗のテーブルを生成します。

```
            NAME        table
main        ; Generate table of powers of 3
value       VAR         1
            REPT        8
            DWORD       value
value       VAR         value * 3
            ENDR
            END         main
```

この例は、次のコードを生成します。

```
     1    000000                  NAME    table
     2    000000           main   ; Generate table of powers of 3
     3    000001           value  VAR     1
     4    000000                  REPT    8
     5    000000                  DWORD   value
     6    000000           value  VAR     value * 3
     7    000000                  ENDR
     7.1  000000 01000000         DWORD   value
     7.2  000003           value  VAR     value * 3
     7.3  000004 03000000         DWORD   value
     7.4  000009           value  VAR     value * 3
     7.5  000008 09000000         DWORD   value
     7.6  00001B           value  VAR     value * 3
     7.7  00000C 1B000000         DWORD   value
     7.8  000051           value  VAR     value * 3
     7.9  000010 51000000         DWORD   value
     7.10 0000F3           value  VAR     value * 3
     7.11 000014 F3000000         DWORD   value
     7.12 0002D9           value  VAR     value * 3
     7.13 000018 D9020000         DWORD   value
     7.14 00088B           value  VAR     value * 3
     7.15 00001C 8B080000         DWORD   value
     7.16 0019A1           value  VAR     value * 3
     8    000020                  END     main
```

ローカル・シンボルの使用法

次の例では、**add1** モジュールに定義されたシンボル **value** は、そのモジュールに対してローカルになります。同じ名前の別のシンボルは、**add2** モジュールに定義されます。

```
          NAME      add1
locn      DEFINE    100H
value     EQU       77
          LDA       locn
          CLC
          ADC       value
          RTS
          ENDMOD

          NAME      add2
value     EQU       88
          LDA       locn
          CLC
          ADCvalue
          RTS
          END
```

モジュール **add1** に定義されるシンボル **locn** は、モジュール **add2** でも利用できます。

ラベルの定義

次の例では、アドレス５のビット番号５にゼロ・ページ・ビット・アドレス指定を与えます。

```
1   000002          value1   EQU    2
2   000003          value2   EQU    3
3   000000 AF05              SEB    value1+value2,value1+value2
4   000002                   END
```

特殊機能レジスタの定義

次の文は、あらゆるアクセスで **P01** を、バイト読み／書きアクセスで **P0** を、そしてバイト書込みアクセスで **P1** を定義します。

```
SFR       P01                      = 02h; port 0 & 1
SFRTYPE   P0       byte,read,write  = 02h ; port 0
SFRTYPE   P1       byte,write       = 03h ; port 1
```

LIMIT 疑似命令の使用

以下の例は、**speed** という変数をセットして、これが 10 から 30 の範囲にあるかどうかをアセンブル時にチェックします。

これは、**speed** がしばしばコンパイル時に変更されるが、しかし定義された範囲を越える値が不定な動作を生じうるような場合に有益です。

```
speed     VAR       23
LIMIT     speed,10,30,"fred out of range"
```

条件付きアセンブリ疑似命令

これらの疑似命令は、ソース・コードの選択アセンブリの論理制御を実行します。

疑似命令	内容
.IF	条件が真ならば、命令をアセンブルする。
.ELSE	条件が偽ならば、命令をアセンブルする。
.ELSEIF	**.IF**... **.ENDIF** ブロックの新規の条件を指定する。
.ENDIF	**.IF** ブロックを終了する。

構文

.IF ***条件***
.ELSE
.ELSEIF ***条件***
.ENDIF

パラメータ

条件　次の１つが適用されます。

絶対式	式には、順方向参照や外部参照があってはいけません。また、ゼロ以外の値は真と見なされます。
文字列１＝文字列２	***文字列１***と***文字列２***の長さとコンテンツが同じ場合、条件は真となります。
文字列１<>文字列２	***文字列１***と***文字列２***の長さとコンテンツが異なっている場合、条件は偽となります。

説明

アセンブリ時にアセンブリ処理を制御するには、**.IF ... .ELSE ... .ENDIF** 疑似命令を使用します。**.IF** 疑似命令に続く条件が真の場合、後続の命令は**.ELSE** か**.ENDIF** 疑似命令が見つけされるまでコードを生成しません（すなわち、アセンブルされたり、構文チェックされたりしません）。

.IF 疑似命令のあとに新規の条件を導入するには、**.ELSEIF** を使用します。

条件付きアセンブラ疑似命令は、アセンブリのどの箇所ででも使用することができますが、マクロ処理と組み合わされた場合に最大限の利用度をもっています。

全てのアセンブラ疑似命令（END を除く）およびファイル組込みは、条件付き疑似命令で不能にすることができます。各.IF 疑似命令は、.ENDIF 疑似命令によって終了してください。.ELSE 疑似命令は任意指定であり、使用された場合は、.IF ... .ENDIF ブロックの中になければなりません。

IF ... .ENDIF および.IF ... .ELSE ... .ENDIF ブロックは、どのレベルでもネストすることができます。

例

次のマクロは、Aに定数を掛け合わせます。

```
          EXTERN    temp
; AX = A * times
mult      MACRO     times
          .IF       times=2
          ASL       A
          TAX                 ; 解答の下位バイト
          LDA       #0
          ROL       A         ; 解答の上位バイト
          .ELSE
          LDM       #times,temp
          LDX       #0
          MUL       temp,X
          TAX                 ; 解答の下位バイト
          PLA                 ; 解答の上位バイト
          .ENDIF
          ENDM
```

マクロの引数が２の場合は、ASL 命令が生成され、命令サイクルを保存します。そうでない場合は、MUL 命令を生成します。

次のプログラムを使用して調べることができます。

```
main      LDA       #17
          mult      2         ; Xに 34 を、Aに0を与える。
          LDA       #23
          mult      13        ; Xに 43 を、Aに1を与える。
          RTS

          END
```

マクロ処理疑似命令

これらの疑似命令を使用すると、ユーザ・マクロを定義することができます。

疑似命令	内容
MACRO	マクロを定義する。
ENDM	マクロ定義を終了する。
EXITM	マクロを途中で終了する。
LOCAL	マクロに対してローカルなシンボルを生成する。
REPT	指定された回数だけ命令をアセンブルする。
REPTC	文字を繰り返し、置換する。
REPTI	文字列を繰返し、置換する。
ENDR	繰返し構造体を終了する。

構文

名前 MACRO [*引数*] ...

ENDM

EXITM

LOCAL *シンボル* [,*シンボル*] ...

REPT *expr*

REPTC *仮*,*実*

REPTC *仮*,*実* [,*実*] ...

ENDR

パラメータ

名前	マクロ名
引数	シンボル引数名
シンボル	マクロに対してローカルとなるシンボル
expr	式
仮	***実***（REPTC）の各文字あるいは各*実*（REPTI）が置換される引数
実	置換される文字列

説明

マクロとは、1つ以上のアセンブラ・ソース行のブロックを表したユーザ定義シンボルのことです。いったんマクロを定義すると、それをちょうどアセンブラ疑似命令やアセンブラ・ニモニックのようにしてプログラムで使用することができます。

アセンブラはマクロに出会うと、マクロの定義を調べ、あたかもマクロによって表される行がソース・ファイルのその位置にあるかのようにそれらの行を挿入します。

マクロは簡単なテキストの置換を効率的に実行しますが、マクロにパラメータを与えることによってマクロの置換作業を制御することができます。

マクロの定義

マクロは、次の文によって定義します。

マクロ名 **MACRO [*arg*] [*arg*] ...**

ここで、マクロ名はマクロに使用する名前を表し、***arg***は、マクロが展開されたときに

マクロに渡したい値の引数を示します。

例えば、マクロ **ERROR** は次のように定義できます。

```
error   MACRO   text
        JSR     abort
        BYTE    text,0
        ENDM
```

これは、パラメータ **text** を使用して、ルーチン **abort** のエラー・メッセージをセットアップします。マクロは、次のような文で呼び出します。

```
        error  'Disk not ready'
```

この文は、アセンブラによって次のように展開されます。

```
        JSR     abort
        BYTE    'Disk not ready',0
```

1つ以上の引数を省略した場合、マクロを呼び出すときに与えられる引数は**\1**～**\9** および**\A**～**\Z** となります。

したがって、上記の例は次のように書き直すことができます。

```
error   MACRO
        JSR     abort
        BYTE    \1,0
        ENDM
```

マクロの途中に出口を生成するには、**EXITM** 疑似命令を使用します。

EXITM は、**REPT ... ENDR**, **REPTC ... ENDR**, **REPTI ... ENDR** の中では使用できません。マクロに対してローカルなシンボルを生成するには、**LOCAL** を使用します。**LOCAL** 疑似命令は、シンボルの使用前に使ってください。

マクロが展開されるたびに、ローカル・シンボルの新しいインスタンスが **LOCAL** 疑似命令によって生成されます。したがって、再帰的マクロの中でのローカル・シンボルの使用は正当です。

マクロを再定義するのは、違法です。

特殊文字を渡す

カンマや空白を含むマクロ引数は、マクロ呼出しの中で突き合わせ引用文字<、>を使用することによって１引数として強制的に解釈させることができます。

例：

```
macl    MACRO   op
        ADC     op
        ENDM
```

これは、次のようにして呼び出すことができます。

```
        macl    <0xf7,x>
        END
```

マクロ引用文字は、**-M** コマンド行オプションによって再定義することができます。このオプションについては、２４ページの「-M」を参照してください。

マクロの処理方法

マクロ処理には、３つの明確な段階があります。

- マクロ定義のスキャンや保存は、アセンブラによって実行されます。**MACRO**、**ENDM** 間のテキストは保存されますが、構文チェックは行われません。インクルード・ファイル参照***$file***は記録され、マクロ展開時に組み込まれます。
- マクロ呼出しは、強制的にアセンブラにマクロ・プロセッサ（エキスパンダ）を呼び出すようにさせます。このエキスパンダは、（まだマクロにない場合に）ソース・ファイルからのアセンブラ入力ストリームをマクロ・エキスパンダからの出力（要求マクロ定義から入力をとります）に切り替えます。

  マクロ・エキスパンダはソース・レベルで置換テキストを取り扱うだけですので、アセンブラ・シンボルの知識は持ち合わせていません。呼び出されたマクロ定義からの行がアセンブラに渡される前に、マクロ・エキスパンダは行をスキャンしてシンボル・マクロ引数の発生をすべて確認し、それを展開引数で置換します。

- 展開された行は、その後他のアセンブラ・ソース行として処理されます。アセンブラへの入力ストリームは、現在のマクロ定義の全ての行が読み取られるまで引き続きマクロ・プロセッサの出力になります。

文の繰返し

命令の同じブロックを多数アセンブルするには、**REPT ... ENDR** 構造体を使用します。***expr*** が0と評価された場合は、何も生成されません。

命令のブロックを文字列の各文字について1回アセンブルするには、**REPTC** を使用します。文字列にカンマがある場合は、引用符で囲んでください。

命令のブロックを一連の文字列の各文字列について1回アセンブルするには、**REPTI** を使用します。文字列にカンマがある場合は、引用符で囲んでください。

例

この項では、アセンブラ・プログラミングがマクロによってより容易になるさまざまな方法例を示します。

命令セットの拡張

マイクロプロセッサ命令セットが展開される方法から、これらの命令セットは必ずしも希望どおりに対称的になっているとは限りません。しかしながら、マクロを記述することによって命令セットに組み込みたいと思う命令に定義を追加し、それらをちょうど組み込み命令のようにして使用することができます。

例えば、740 はAレジスタとBレジスタ間の転送を行うための TSA 命令や TAS 命令を組み込んでいません。これを実行するために、次のように **tsa**、**tas** マクロを定義することができます。

```
tsa     MACRO
        TSX
        TXA
        ENDM
tas     MACRO
        TAX
        TXS
        ENDM
```

これは、プログラムで次のように使用することができます。

```
          EXTERN    value,routine
point_at_stack
          tsa
          CLC
          ADC       value
          TAX                 ; スタックがページ０と想定する変数のポイント
                              ; インライン・コード化による効率向上
          JSR       routine
          RTS
          END
```

時間が重要なコードでは、ルーチンをインラインでコード化してサブルーチン呼出しやリターンのオーバーヘッドを回避させることが望ましい場合があります。マクロは、これを行う便利な方法を提供してくれます。

例えば、次のサブルーチンはバッファからのバイトをポートに出力します。

```
          NAME      play
          EXTERN    port

          RSEG      RAM
buffer    BLKB      256

          RSEG      PROM
play      LDX       #0
loop      LDA       buffer,X
          STA       port
          INX
          BNE       loop
          RTS
```

メイン・プログラムは、このルーチンを次のように呼び出します。

```
doplay    JSR       play
          RTS

          END
```

効率を得るために、これを次のマクロのように再コード化します。

```
            NAME        play
            EXTERN      port

            RSEG        RAM
buffer      BLKB        256

play        MACRO
            LOCAL       loop
            LDX         #0
loop        LDA         buffer,X
            STA         port
            INX
            BNE         loop
            ENDM
```

ラベル **loop** をマクロに対してローカルにするには、**LOCAL** を使用します。そうしなければ、**loop** ラベルが既に存在するために、マクロが 2 回使用されるとエラーが発生します。 インライン・コードを使用するには、単にメイン・プログラムを次のように変更します。

```
            RSEG        PROM
doplay      play
            RTS

            END
```

REPTC、REPTI の使用法

次の例は、サブルーチン **PLOT** に対する一連の呼出しをアセンブルし、文字列の各文字をプロットします。

```
            NAME        reptc

            EXTERN      plotc
banner      REPTC       chr, "Welcome"
            LDA         'chr'
            JSR         plotc
            ENDR

            END
```

これは、次のコードを生成します。

```
     1    000000                    NAME    reptc
     2    000000
     3    000000                    EXTERN  plotc
     4    000000          banner    REPTC   chr,"Welcome"
     5    000000                    LDA     'chr'
     6    000000                    JSR     plotc
     7    000000                    ENDR
     7.1  000000 A557               LDA     'W'
     7.2  000002 20....             JSR     plotc
     7.3  000005 A565               LDA     'e'
     7.4  000007 20....             JSR     plotc
     7.5  00000A A56C               LDA     'l'
     7.6  00000C 20....             JSR     plotc
     7.7  00000F A563               LDA     'c'
     7.8  000011 20....             JSR     plotc
     7.9  000014 A56F               LDA     'o'
     7.10 000016 20....             JSR     plotc
     7.11 000019 A56D               LDA     'm'
     7.12 00001B 20....             JSR     plotc
     7.13 00001E A565               LDA     'e'
     7.14 000020 20....             JSR     plotc
     8    000023
     9    000023                    END
```

次の例では、**REPTI** を使用して多数のメモリ・ロケーションをクリアします。

```
          NAME      repti

          EXTERN    base,count,init
clear     REPTI     adds,base,count,init
          LDM       #0,adds
          ENDR

          END
```

これは、次のコードを生成します。

```
1    000000                  NAME    repti
2    000000
3    000000                  EXTERN  base,count,init
4    000000           clear  REPTI   adds,base,count,init
5    000000                  LDM     #0,adds
6    000000                  ENDR
6.1  000000 3C00..           LDM     #0,base
6.2  000003 3C00..           LDM     #0,count
6.3  000006 3C00..           LDM     #0,init
7    000009
8    000009                  END
```

構造化アセンブリ疑似命令

構造化疑似命令を使用すると、ループや制御構造体をアセンブリ・レベルで実行させることができます。

疑似命令	内容
IF	条件が真ならば、実行する命令を指定する。
ELSE	条件が偽ならば、実行する命令を導入する。
ELSEIF	IF ブロックで新しい条件を指定する。
ENDIF (ENDI,THEN)	IF ブロックを終了する。
WHILE	条件が真の間、後続の命令を繰り返す。
ENDW (WEND)	WHILE ループを終了する。
REPEAT	条件が真になるまで、後続の命令を繰り返す。
UNTIL	REPEAT ループを終了する。
FOR	後続の命令を指定回数だけ繰り返す。
ENDF	FOR ループを終了する。
SWITCH	複数の大文字／小文字の切換
CASE	SWITCH ブロックの大文字／小文字
DEFAULT	デフォルトの SWITCH ブロックの大文字／小文字
ENDS	SWITCH ブロックを終了する。
BREAK	ループや分岐構成体を途中で終了する。
CONTINUE	ループや分岐構成体の実行を継続する。

構文

```
IF {条件 | 式}
ELSE
ELSEIF {条件 | 式}
ENDIF

WHILE {条件 | 式}
ENDW

REPEAT
UNTIL {条件 | 式}

FOR reg = start {TO | DOWNTO} end {BY |STEP} step
ENDF

SWITCH reg
CASE op [.. op2]
DEFAULT
ENDS

BREAK レベル
CONTINUE
```

パラメータ

条件	次の条件のいずれか１つ	
	<CC> キャリー・クリア	**<CS>** キャリー・セット
	<EQ> 等しい	**<NE>** 等しくない
	<PL> プラス	**<MI>** マイナス
	<VC> オーバフロー・クリア	**<VS>** オーバフロー・セット
式	次の形式をもつ式	
	reg rel op	
reg	次のレジスタのいずれか１つ	
	Ａ、Ｘ、またはＹ	
rel	次の関係のいずれか１つ	
	>=, <=, !=, <>, ==, =, >, または<	
op, op1, op2	***reg***によって異なる **CMP**, **CPX**, **CPY** の有効なアドレス指定モード	

start, ***end***, ***step*** 下表に指定されたアドレス指定モードのいずれか

Reg	*Start*	*End*	*Step*
A	即値 ゼロページ ゼロページX 絶対 絶対X 絶対Y ゼロページ間接X ゼロページ間接Y	即値 ゼロページ ゼロページX 絶対 絶対X 絶対Y ゼロページ間接X ゼロページ間接Y	即値 ゼロページ ゼロページX 絶対 絶対X 絶対Y ゼロページ間接X ゼロページ間接Y
X	即値 ゼロページ ゼロページY 絶対 絶対Y	即値 ゼロページ 絶対	1または-1に制限
Y	即値 ゼロページ ゼロページX 絶対 絶対X	即値 ゼロページ 絶対	1または-1に制限

アドレス指定モードの説明については、 123ページの「アセンブラ命令」を参照してください。

step が省略された場合に **DOWNTO** が指定されていると、デフォルト値は#1あるいは#-1となります。この構造体のインクレメント、デクレメントについては、***reg*** がAの場合は、（**INC** ではなく）**ADC/SBC** によってインクレメントします。したがって、ステップ値を選択する場合に自由度が広がります。

レベル ブレークするレベルの数、1～3

説明

IAR アセンブラは、アセンブリ・レベルでより簡単にループや制御構造体を実現するために広範な疑似命令を構造化アセンブリに用意しています。

構造化アセンブリ疑似命令を使用する利点は、得られるプログラムがよりクリアになり、その論理がより理解しやすくなる点にあります。

疑似命令は、結果のプログラムがマニュアル的にプログラムされたかのように効率的に単純かつ予測可能なコードを生成するように作成されています。

条件付き構成体

比較命令やジャンプ命令用にアセンブラ・ソース・コードを生成するには、**IF ... ENDIF** を使用します。生成されたコードは通常のコードのようにアセンブルすることができ、マクロに似ています。このコードを、条件付きアセンブリと混同しないでください。

IF ブロックは、どのレベルにおいても入れ子にすることができます。

IF *条件*の形式は、次のように文を生成します。

```
B??    else_label
```

ここで、??は逆の条件です。すなわち、**IF CC** は **BCS** を与えます。

IF 式の形式は、次のようにコードを生成します。

```
CMP op
B??    else_label
```

ここで、**CMP** 命令は、***op*** に定義された比較命令であればどれでも構いません。また、分岐は次のように生成されます。

関係	分岐ラベル		
>=		BMI	else_label
<=		BEQ	less_eq
		BPL	else_label
	less_eq		
!= or < >		BEQ	else_label
= or =		BNE	else_label
>		BMI	else_label
		BEQ	else_label
<		BPL	else_label

IF 条件が偽の場合に実行される命令を導入するには、**IF** 疑似命令の後に **ELSE** を使用します。

IF 疑似命令の後に新しい条件を導入するには、**ELSEIF** を使用します。

ループ疑似命令

式が **TRUE** である限り実行されるループの生成には、**WHILE ... ENDW** を使用します。式がループの始まりで偽の場合、ループ本体は実行されません。

式が **FALSE** である限り少なくとも１回は実行される本体をもつループの生成には、**REPEAT ... UNTIL** 構成体を使用します。

WHILE ... ENDW ループ、**REPEAT ... UNTIL** ループ、あるいは **CONTINUE** を途中で終了し、ループの次の反復を継続するには、**BREAK** を使用します。

疑似命令は、IF 疑似命令と同じ文を生成します。

反復構成体

命令をアセンブルして、指定されたシーケンス値だけ命令ブロックを繰り返すには、**FOR** を使用します。

reg がＡの場合、**FOR** ループは次のコードを生成します。

文	生成されたコード
FOR A = starts,X TO #3 BY $3	**LDA starts,X**
	BRA skip_label
	loop_label:
ENDF	**cont_label; CLC**
	ADC $3
	skip_label: CMP #3
	BMI LOOP_LABEL
	BEQ loop_label
	break_label:

FOR ループを途中で終了し、**ENDF** に続く命令を継続して実行するには、**BREAK** を使用します。

ループの次の繰返しを継続するには、**CONTINUE** が使用できます。

分岐構成体

テストの値に応じて、多数の文の中の１つを実行するには、**SWITCH ... ENDS** ブロックを使用します。

CASE は各テストを定義し、**DEFAULT** は常に真となる **CASE** を導入します。

CASE は、Ｃの場合のようにデフォルトでは実現されません。**CASEIN** は、***op1*** が ***op2*** 未満、あるいはそれと同等でなければならない場合に範囲のチェックを行うという点を除けば **CASE** と似ています。

BREAK は、**SWITCH ... ENDS** ブロックの終了に使用することができます。

SWITCH 構成体は、次のコードを生成します。

文	生成コード		
SWITCH REG	none		
CASE op		BRA	skip_label
(no preceding BREAK)	link_label(n)		
		CMP	op
		BNE	link_label (n+1)
	skip_label		
SWITCH reg	none		
CASE op	link_label(n)		
(preceding BREAK)		CMP	op
		BNE	link_label (n+1)
CASE op1 ... op 2	link_label(n)		
		CMP	op1
		BCC	link_label (n+1)
		CMP	op2
		BEQ	equal_op2
		BCS	link_label (n+1)
	equals_op2		
DEFALT	link_label		
ENDS	link_label		
	break_label		

例

条件付き構成体の使用法

次のプログラム例はＡレジスタを調べ、レジスタの値が０未満、０、０より大きいいずれかに応じて'N', 'Z'、'P'をプロットします。

```
zero:
          NAME      else
          EXTERN    plot
main      IF        A < 0
```

```
        LDA     #'N'
        ELSEIF  A = 0
        LDA     #'Z'
        ELSE
        LDA     #'P'
        ENDIF
        JSR     plot
        RTS

        END     main
```

これは、次のコードを生成します。

```
 1   000000         zero:
 2   000000                NAME   else
 3   000000                EXTERN plot
 4   000000         main   IF     A < 0
4.1  000000 C500           CMP    0
4.2  000002 B004           BCS    _?0
 5   000004 A94E           LDA    #'N'
 6   000006                ELSEIF A = 0
6.1  000006 800A           BRA    _?1
6.2  000008         _?0
6.3  000008 C500           CMP    0
6.4  00000A D004           BNE    _?2
 7   00000C A95A           LDA    #'Z'
 8   00000E                ELSE
8.1  00000E 8002           BRA    _?1
8.2  000010         _?2
 9   000010 A950           LDA    #'P'
10   000012                ENDIF
10.1 000012         _?1
11   000012 20....         JSR    plot
12   000015 60             RTS
13   000016
14   000016                END    main
```

ループ構成体の使用法

次の例は **REPEAT ... UNTIL** ループを使用してレジスタAのビット順序を逆にし、結果を変数 **hibyte** に置きます。

```
          NAME      repeat
          EXTERN    hibyte
reverse   REPEAT
          LSR       A
          ROL       hibyte
          UNTIL     A <> #0
          RTS

          END
```

これは、次のコードを生成します。

```
     1    000000                  NAME    repeat
     2    000000                  EXTERN  hibyte
     3    000000          reverse REPEAT
     3.1  000000          _?0
     4    000000 4A               LSR     A
     5    000001 26..             ROL     hibyte
     6    000003                  UNTIL   A <> #0
     6.1  000003 C900             CMP     #0
     6.2  000005 F0F9             BEQ     _?0
     6.3  000007          _?1
     7    000007 60               RTS
     8    000008
     9    000008                  END
```

FOR... NEXT の使用法

次の例は、FOR ブロックを使用して 100 値バッファをポートに出力します。

```
          NAME      for
          EXTERN    port,buffer
play      FOR       X = #0 TO #100 STEP #1
          LDA       buffer,X
          STA       port
          ENDF      A <> #0
          RTS

          END
```

これは、次のコードを生成します。

```
1    000000                  NAME    for
2    000000                  EXTERN  port,buffer
3    000000          play    FOR     X = #0 TO #100 STEP #1
3.1  000000 A200             LDX     #0
3.2  000002 8005             BRA     _?1
3.3  000004          _?0
4    000004 B5..             LDA     buffer,X
5    000006 85..             STA     port
6    000008                  ENDF    A <> #0
6.1  000008 E8       _?2     INX
6.2  000009 E064     _?1     CPX     #100
6.3  00000B F0F7             BEQ     _?0
6.4  00000D 90F5             BCC     _?0
6.5  00000F          _?3
7    00000F 60               RTS
8    000010
9    000010                  END
```

分岐構成体の使用法

次の例は **SWITCH ... ENDS** ブロックを使用し、Aレジスタの値に応じて **Zero**、**Positive**、あるいは **Negative** を出力します。このブロックは、外部 **print** ルーチンを使用して即値文字列を出力します。

```
          NAME      switch
          EXTERN    print
test      SWITCH    A

          CASE      #0
          JSR       print
          BYTE      "Zero",0
          BREAK

          CASE      #$8 .. #$FF
          JSR       print
          BYTE      "Negative",0
          BREAK

          DEFAULT
          JSR       print
          BYTE      "Positive",0

          ENDS
```

```
        END       test
```

これは、次のコードを生成します。

```
 1    000000                    NAME    switch
 2    000000                    EXTERN  print
 3    000000          test      SWITCH  A
 4    000000
 5    000000                    CASE    #0
 5.1  000000 C900               CMP     #0
 5.2  000002 D00B               BNE     _?1
 6    000004 20....             JSR     print
 7    000007 5A65726F*          BYTE    "Zero",0
 8    00000D                    BREAK
 8.1  00000D 8026               BRA     _?0
 9    00000F
10    00000F                    CASE    #$8 .. #$FF
10.1  00000F C908       _?1     CMP     #$8
10.2  000011 9015               BCC     _?3
10.3  000013 C9FF               CMP     #$FF
10.4  000015 F002               BEQ     _?2
10.5  000017 B00F               BCS     _?3
10.6  000019            _?2
11    000019 20....             JSR     print
12    00001C 4E656761*          BYTE    "Negative",0
13    000026                    BREAK
13.1  000026 800D               BRA     _?0
14    000028
15    000028                    DEFAULT
15.1  000028            _?3
16    000028 20....             JSR     print
17    00002B 506F7369*          BYTE    "Positive",0
18    000035
19    000035                    ENDS
19.1  000035            _?0
20    000035                    END     test
```

リスト制御疑似命令

これらの疑似命令は、アセンブラ・リストを制御します。

疑似命令	内容
LSTCND	条件付きアセンブリ・リストを制御する。
LSTCOD	マルチ行コード・リストを制御する。
LSTCYC	サイクル・カウントのリストを制御する。
LSTEXP	マクロで生成された行のリストを制御する。
LSTMAC	マクロ定義のリストを制御する。
LSTOUT	アセンブリ・リスト出力を制御する。
LSTPAG	ページへの出力形式を制御する。
LSTREP	繰返し疑似命令が作成した行のリストを制御する。
LSTSAS	構造化アセンブリ・リストを制御する。
LSTXRF	相互参照テーブルを生成する。
PAGSIZ	各ページの行数を設定する。
COL	各ページのカラム数を設定する。
PAGE	新しいページを生成する。
CYCLES	リストされたサイクル・カウントを設定する。
CYCMAX	各命令に最大サイクル・カウントを使用する。
CYCMEAN	各命令に平均サイクル・カウントを使用する。
CYCMIN	各命令に最小サイクル・カウントを使用する。

構文

LSTCND{+ | –}

LSTCOD{+ | –}

LSTCYC{+ | –}

LSTEXP{+ | –}

LSTMAC{+ | –}

LSTOUT{+ | –}

LSTPAG{+ | –}

LSTREP{+ | –}

LSTSAS{+ | –}

LSTXRF{+ | –}

COL ***カラム***

PAGSIZ ***行***

PAGE

CYCLES ***サイクル***

CYCMAX

CYCMEAN

CYCMIN

パラメータ

カラム 80～132 の範囲の絶対式であり、デフォルト値は 132。

行 10～150 の範囲の絶対式

サイクル リストされたサイクル・カウントが設定される値

説明

リストのオン、オフ

エラー・メッセージを除く全てのリストの出力を不能にするには、**LISTOUT-**を使用します。これは、その他のリスト制御疑似命令をすべて無効にします。

デフォルトは、(リスト・ファイルが指定されている場合に) 出力をリストする **LISOUT+**です。

条件付きコードと文字列のリスト

前回の条件付き**.IF** 文、**.ELSE**、あるいは **END** によって不能化されていないアセンブリの部分についてのみ、強制的にアセンブラにソース・コードをリストさせるには、**LSTCND+**を使用します。

デフォルトの設定は、全ソース行をリストする **LSTCND-**です。

必要に応じて出力コードのリストを２行以上に拡張するには、**LSTCOD+**を使用します。

つまり、長 ASCII 文字列は、リスト出力を数行生成します。

デフォルトの設定は、ソース行に最初のコード行のみをリストする **LSTCOD-**です。コードの生成は、影響を受けません。

マクロ・リストの制御

マクロ生成行のリストを不能にするには、**LSTEXP-**を使用します。デフォルトは、全てのマクロ生成行をリストする **LSTEXP+**です。

マクロ定義をリストするには、**LSTMAC+**を使用します。デフォルトは、マクロ定義のリストを不能にする **LSTMAC-**です。

生成された行のリストの制御

REPT, REPTC, REPTI 疑似命令で生成された行のリストをオフにするには、**LSTREP-**を使用します。

デフォルトは、生成された行をリストする **LSTREP+**です。

構造化アセンブリの疑似命令で生成されたアセンブリ・ソースのリストを不能にするには **LSTSAS-**を使用します。

デフォルトは、構造化アセンブリ疑似命令で生成されたアセンブリ・ソースをリストする **LSTSAS+**です。

相互参照テーブルの生成

カレント・モジュールのアセンブリ・リストの終りに相互参照テーブルを生成するには、**LSTXRF+**を使用します。相互参照テーブルには、値、行番号、シンボルの型が示されます。

デフォルトは、相互参照テーブルを生成しない **LSTXRF-**です。

リストされた出力の書式化

アセンブリ・リストのページ毎のカラム数を設定するには、**COL** を使用します。デフォルトのカラム数は、132 です。

アセンブリ・リストのページ毎の出力行数を設定するには、**PAGSIZ** を使用します。デフォルトのページ毎の行数は、44 です。

アセンブリ出力リストをページに書式化するには、**LSTPAG+**を使用します。

デフォルトは、連続したリストを出力する **LSTPAG-**です。

ページングが実行可能状態にある場合にアセンブリ・リストに新しいページを生成するには、**PAGE** を使用します。

サイクルのカウント

サイクル・カウントをリストするには、**LSTCYC+**を使用します。表示値は、プロセッサ・クロック・サイクルの合計値で、この合計値は **CYCLES** 疑似命令で希望の値にリセットすることができます。

表示されたサイクル・カウントは、キャッシュ、サイクル・オーバーラップ、あるいはその他のランタイム依存性のないものと想定されることに注意してください。

コードの一部でサイクル・カウントを計算する場合のカウント設定には、**CYCLES** を使用します。

サイクル・カウントは、リストの始まりで 0 に設定されます。

サイクル・カウントを出力するには、**LSTCYC+**疑似命令、つまりコマンド行**-cC** オプションを使用してください。

異なったサイクル・カウントをもつ命令に最大、平均、最小のサイクル・カウントを使用するには、**CYCMAX** や **CYCMEAN** や **CYCMIN** を使用します。

例

リストのオン／オフ

プログラムのデバッグ・セクションのリストを不能にするには、

```
LSTOUT-
; デバッグされたセクション
LSTOUT+
; まだデバッグされていない
```

条件付きコードと文字列のリスト

次の例は、**LSTCND+**が、IF 疑似命令によって不能にされたサブルーチンに呼出しを隠す方法を示します。

```
          NAME      lstcnd
          EXTERN    print

debug     VAR       0

begin     .IF       debug
          JSR       print
          .ENDIF

          LSTCND+
begin2    .IF       debug
          JSR       print
          .ENDIF

          END       begin
```

これは、次のリストを生成します。

```
     1    000000                    NAME    lstcnd
     2    000000                    EXTERN  print
     3    000000
     4    000000          debug     VAR     0
     5    000000
     6    000000          begin     .IF     debug
     7    000000                    JSR     print
     8    000000                    .ENDIF
     9    000000
    10    000000                    LSTCND+
    11    000000          begin2    .IF     debug
    13    000000                    .ENDIF
    14    000000
    15    000000                    END     begin
```

次の例は、WORD 疑似命令で生成されたコードでの LSTCOD+の効果を示します。

```
1 000000                          NAME       lstcod
2 000000 01000A00  table          WORD       1,10,100,1000,10000
3 00000A
4 00000A                          LSTCOD+
5 00000A 01000A00  table2         WORD       1,10,100,1000,10000
         6400E803
         1027
6 000014                          END
```

マクロのリストの制御

次の例は、LSTMAC、LSTEXP の効果を示します。

```
          NAME       lstmac

          EXTERN     memloc
times2    MACRO      addr
          ASL        addr
          ENDM

          LSTMAC+
div2      MACRO      addr
          LSR        addr
          ENDM
```

```
begin     times2    memloc

          LSTEXP-
          div2      memloc

          END       begin
```

これは、次の出力を生成します。

```
     1    000000                NAME    lstmac
     2    000000
     3    000000                EXTERN  memloc
     7    000000
     8    000000                LSTMAC+
     9    000000          div2  MACRO   addr
    10    000000                LSR     addr
    11    000000                ENDM
    12    000000
    13    000000          begin times2  memloc
   13.1   000000 06..           ASL     memloc
   13.2   000002                ENDM
    14    000002
    15    000002                LSTEXP-
    16    000002                div2    memloc
    17    000004
    18    000004                END     begin
```

生成された行のリストの制御

次の例は、**LSTSAS-**の効果を示します。

```
          NAME      lstsas

begin     IF        A < 7
          ROL       A
          ENDIF

          LSTSAS-
          IF        A < 7
          ROL       A
          ENDIF

          END       begin
```

これは、次のリストを生成します。

```
     1    000000                NAME    lstsas
```

```
      2    000000
      3    000000          begin   IF      A < 7
    3.1    000000 C507             CMP     7
    3.2    000002 B001             BCS     _?0
      4    000004 2A               ROL     A
      5    000005                  ENDIF
    5.1    000005          _?0
      6    000005
      7    000005                  LSTSAS-
      8    000005                  IF      A < 7
      9    000009 2A               ROL     A
     10    00000A                  ENDIF
     11    00000A
     12    00000A                  END     begin
```

次の例は、LSTREP-の効果を示します。

```
          NAME      tables

main      ; 3 の累乗のテーブルの生成
calc      VAR       1
          REPT      4
          WORD      calc
calc      VAR       calc*3
          ENDR

          LSTREP-
          ; 7 の累乗のテーブルの生成
calc      VAR       1
          REPT      4
          WORD      calc
calc      VAR       calc*7
          ENDR

          END       main
```

これは、次のリストを生成します。

```
 1    000000                  NAME    tables
 2    000000
 3    000000          main    ; 3 の累乗のテーブルの生成
 4    000001          calc    VAR     1
 5    000000                  REPT    4
 6    000000                  WORD    calc
 7    000000          calc    VAR     calc*3
 8    000000                  ENDR
8.1   000000 0100             WORD    calc
8.2   000003          calc    VAR     calc*3
8.3   000002 0300             WORD    calc
8.4   000009          calc    VAR     calc*3
8.5   000004 0900             WORD    calc
8.6   00001B          calc    VAR     calc*3
8.7   000006 1B00             WORD    calc
8.8   000051          calc    VAR     calc*3
 9    000008
10    000008                  LSTREP-
11    000008                  ; 7 の累乗のテーブルの生成
12    000001          calc    VAR     1
13    000008                  REPT    4
14    000008                  WORD    calc
15    000008          calc    VAR     calc*7
16    000008                  ENDR
17    000010
18    000010                  END     main
```

リストされた出力の書式化

次の例は、出力をそれぞれが 80 カラムある 66 行のページに書式化します。**PAGE** 疑似命令は、モジュール間にページ・ブレークを挿入します。

```
PAGSIZ 66 ; ページ・サイズ
COL 80
LSTPAG+
...
ENDMOD
PAGE
MODULE
...
```

サイクル・カウントの表示

次の例は、**CYCLES 0** を使用して取込みルーチン内のメイン・ループのサイクル・カウントを計算します。サイクル・カウントは、**LSTCYC+**を使用してリスト内に表示されます。**CYCMEAN** は、各命令について平均サイクル・カウントが計算されるようにします。相互参照リストを与えるには、**LSTXREF+**を使用します。

```
     1              000000                  NAME    cycles
     2              000000                  LSTCYC+
     3            0 000000                  LSTXRF+
     4            0 000000                  EXTERN  ioport
     5            0 000000
     6            0 000000                  RSEG    ram
     7            0 0000C8          size    VAR     200
     8            0 000000          buffer  BLKB    size
     9            0 0000C8
    10            0 000000                  RSEG    rom
    11            0 000000                  CYCLES  0
    12            0 000000                  CYCMEAN
    13            0 000000 A200     capture LDX     #0
    14            2 000002 A5..     capt2   LDA     ioport
    15            6 000004 95..             STA     buffer,X
    16           11 000006 E8               INX
    17           13 000007 E0C8             CPX     #size
    18           15 000009 D0F7             BNE     capt2
    19           18 00000B 60               RTS
    20           24 00000C
    21           24 00000C                  END


Segment             Type        Mode
----------------------------------------
ram                 UNTYPED     REL
rom                 UNTYPED     REL

Label               Mode   Type                    Segment    Value/Offset
------------------------------------------------------------------------------
buffer              REL    CONST UNTYP.            ram        0
capt2               REL    CONST UNTYP.            rom        2
capture             REL    CONST UNTYP.            rom        0
ioport              ABS    CONST EXT [000] UNTYP.  __EXTERNS  Solved Extern
size                ABS    VAR UNTYP.              ram        C8
```

C形式プリプロセッサ疑似命令

次のC言語プリプロセッサ疑似命令が利用できます。

疑似命令	内容
#define	ラベルに値を割り付ける。
#undef	ラベルを未定義にする。
#if	条件が真ならば、命令をアセンブルする。
#ifdef	シンボルが定義されると、命令をアセンブルする。
#ifndef	シンボルが未定義ならば、命令をアセンブルする。
#else	条件が偽ならば、命令をアセンブルする。
#endif	**#if, #ifdef**、あるいは**#ifndef** ブロックを終了する。
#include	ファイルを組み込む。
#error	エラーを生成する。

構文

#define ***ラベル テキスト***

#undef ***ラベル***

#if ***条件***

#ifdef ***ラベル***

#ifndef ***ラベル***

#else

#endif

#include {"***filename***" | <***filename***>}

#error "***メッセージ***"

パラメータ

ラベル 定義、未定義、あるいはテストするシンボル

テキスト 割り付ける値

条件	次のいずれか１つ	
	絶対式	式には、順方向参照や外部参照があってはいけません。また、ゼロ以外の値は真と見なされます。
	文字列１＝文字列２	***文字列１***と***文字列２***の長さとコンテンツが同じ場合、条件は真となります。
	文字列１<>文字列２	***文字列１***と***文字列２***の長さとコンテンツが異なる場合、条件は真となります。
filename	組み込むファイルの名前	
メッセージ	表示するテキスト	

説明

ラベルの定義と未定義

一時的ラベルを定義するには、**#define** を使用します。

#define ***ラベル 値***

は、次と同じです。

ラベル **VAR** ***値***

ラベルを未定義にするには、**#undef** を使用します。その効果は、ラベルが定義されていないかのようにします。

条件付き疑似命令

アセンブリ処理をアセンブリ時に制御するには、**#if ... #else ... #endif** 疑似命令を使用します。**#if** 疑似命令に続く条件が真でない場合、後続の命令は**#endif** あるいは**#else** 疑似命令が検出されるまで、コードを生成しません（すなわち、アセンブルされたり、構文チェックされたりしません）。

全てのアセンブラ疑似命令（**END** を除く）およびファイル組込みは、条件付き疑似命令で不能にすることができます。各**#if** 疑似命令は、**#endif** 疑似命令で終了してください。**#else** 疑似命令は任意指定です。また使用する場合は、**#if ... #endif** ブロック内にいれてください。

#if ... #endif ブロック、#if ... #else ... #endif ブロックは、どのレベルにおいていも入れ子にすることができます。

シンボルが定義された場合に限り命令を次の**#else** あるいは**#endif** 疑似命令までアセンブルするには、**#ifdef** を使用します。

シンボルが未定義の場合に限り命令を次の**#else** あるいは**#endif** 疑似命令までアセンブルするには、**#ifndef** を使用します。

ソース・ファイルの組込み

ファイルのコンテンツを、指定された点のソース・ファイルに挿入するには、**#include** を使用します。

エラーの表示

ユーザ定義テストの場合などで強制的にアセンブラにエラーを発生させるには、**#error** を使用します。

例

条件付き疑似命令の使用法

次の例はラベル **adjust** を定義し、その後条件付き疑似命令**#ifdef** を使用して、値が定義されている場合はその値を適用します。値が定義されていない場合は、**#error** がエラーを表示します。

```
          NAME      ifdef
          EXTERN    input,output
#define   adjust    10

main      LDA       input
          CLC
#ifdef    adjust
          ADC       #adjust
#else
#error    " 'adjust' not defined"
#endif
#undef    adjust
          STA       output
          RTS

          END
```

ソース・ファイルの組込み

次の例は、**#include** を使用して、マクロ定義したファイルをソース・ファイルに組み込みます。例えば、次のマクロは **macros.s31** に定義されます。

```
exch        MACRO       add1,add2
            PHA
            LDA         add1
            PHA
            LDA         add2
            STA         add1
            PLA
            STA         add2
            PLA
            ENDM
```

その後**#include** を使用して、次の例のようにマクロ定義が組み込まれます。

```
            NAME        include
            ; 標準マクロの定義
#include “macros.s31”
            ; プログラム
            EXTERN      var1,var2
main        exch        var1,var2
            RTS
            END         main
```

データ定義／割当疑似命令

これらの疑似命令は、一時値や予約メモリを定義します。

疑似命令	内容
BLKB	８ビット・バイトにスペースを割り当てる。
BLKW	１６ビット・ワードにスペースを割り当てる。
BLKD	３２ビット倍長語にスペースを割り当てる。
BYTE	８ビット・バイトを生成する。
WORD	１６ビット・ワードを生成する。
DWORD	３２ビット・ワードを生成する。

構文

LKB ***expr***
BLKW ***expr***
BLKD ***expr***
BYTE ***expr*****[,*****expr*****]**
WORD ***expr*****[,*****expr*****]**
DWORD ***expr*****[,*****expr*****]**

パラメータ

expr 有効な絶対式、再配置可能な式、外部式、あるいは ASCII 文字列です。ASCII 文字列では、サイズの倍数に 0 が入れられます。

説明

メモリ・スペースを初期化、確保するには、**BYTE**、**WORD**、**DWORD** を使用します。

スペースを割り当てるには、**BLKB**、**BLKW**、**BLKD** を使用します。メモリ・コンテンツは、どのような方法によっても初期化することはできません。

例

次の例は、ルーチンにアドレス参照テーブルを生成します。

```
          NAME      table
          EXTERN    operand

table     WORD      addsubr,subsubr,clrsubr

addsubr   ADC       operand
          RTS

subsubr   SBC       operand
          RTS

clrsubr   LDA       #0
          RTS

          END
```

文字列の定義

文字列を定義するには、

```
message BYTE 'Please enter your name'
```

文字列に引用符を入れるには、２回入力します。例えば、次のとおりです。

```
error BYTE 'Dont''t understand!'
```

0xA バイトにスペースを確保するには、

```
table     BLKB      0xA
```

アセンブラ・コントロール疑似命令

これらの疑似命令は、アセンブラの操作を制御します。

疑似命令	内容
$(INCLUDE)	ファイルを組み込む。
/* コメント ***/**	C形式コメント・デリミタ
RADIX	デフォルトのベースを設定する。
CASEON	大文字／小文字の区別を可能にする。
CASEOFF	大文字／小文字の区別を不能にする。

構文

$*filename*

/* *コメント* ***/**

RADIX *expr*

CASEON

CASEOFF

パラメータ

filename 組み込むファイルの名前。行の最初の文字には、**$**文字を使用してください。

コメント アセンブラでは無視されるコメント

expr デフォルトのベース。省略値は１０（１０進法）です。

説明

ファイルのコンテンツを、指定された点のソース・ファイルに挿入するには、**$**を使用します。

アセンブラ・リストのセクションをコメントするには、**/*...*/**を使用します。

デフォルトのベースを設定して ASCII ソースから内部２進形式への定数変換に使用するには、**RADIX** を用います。

ベースを 16 から 10 へリセットするには、***expr*** を１６進数で、あるいは指定されたナンバー・ベースで記述してください。例えば、次のとおりです。

```
RADIX A
RADIX 0x0A
RADIX D'10
RADIX B'1010
```

大文字／小文字の区別の制御

ユーザ定義シンボルの大文字／小文字の区別をオン／オフするには、**CASEON** ある **CASEOFF** を使用します。デフォルトでは、大文字／小文字の区別はオフになります。

CASEOFF がアクティブな場合、シンボルはすべて大文字で格納されます。XLINK で使用されるシンボルはすべて、XLINK 定義ファイルで大文字で記述してください。

例

ソース・ファイルの組込み

次の例は、**$**を使用してマクロ定義ファイルをソース・ファイルに組み込みます。例えば、次のマクロは **macros.s31** に定義されます。

```
; メモリ交換
exch        MACRO       add1,add2
            PHA
            LDA         add1
            PHA
            LDA         add2
            STA         add1
            PLA
            STA         add2
            PLA
            ENDM
```

マクロ定義は、次のように＄疑似命令で組**み込むことができます。**

```
            NAME        include
            ; 標準のマクロ定義
$macros.s31
            ; プログラム
            EXTERN      var1,var2
main        exch        var1,var2
            RTS
            END         main
```

コメントの定義

次の例は、マルチ行コメントで/* ... */を使用する方法を示しています。

```
/*
program to read serial input.
Version 2: 19.6.94
Author: mjp
*/
```

ベースの変更

デフォルトのベースを１６に設定するには、

```
RADIX D'16
LDA   #12
```

すると、即値引数は H'12 として解釈されます。

大文字／小文字の区別の制御

デフォルトでは、CASEOFF が使用可能な状態になっていますので、次の例の label と LABEL は同じと見なされます。

```
label   NOP       ;"LABEL"として格納される。
        JMP       LABEL
```

しかしながら、次の例は重複ラベル・エラーを発生します。

```
label   NOP
LABEL   NOP       ;エラー："LABEL"は既に定義されています。
        END
```

第８章　アセンブラ命令

この章では、740 命令を一覧します。

アドレス指定モード

命令には、次のモードでアドレス指定されたオペランドを含むことができます。

アドレス指定モード	オペランド構文	例
暗示		
即値	**#nn**	**ADC #$A5**
アキュムレータ	**A**	**ASL A**
ゼロページ	**Zz**	**ADC $02**
ゼロページＸ	**zz, X**	**ADC $02, X**
ゼロページＹ	**zz, Y**	**LDX $02, Y**
ゼロページ間接	**(zz)**	**JMP ($05)**
ゼロページ間接Ｘ	**(zz, X)**	**ADC ($1E, X)**
ゼロページ間接Ｙ	**(zz), Y**	**ADC ($1E), Y**
絶対	**hhll**	**ADC $AD12**
絶対Ｘ	**hhll, X**	**ADC $AD12, X**
絶対Ｙ	**hhll, Y**	**ADC $AD12, Y**
絶対間接	**(hhll)**	**JMP ($1400)**
特殊ページ	**\hhll**	**JSR \$FFE0**
相対	**hhll**	**BCC *-12**
アキュムレータ・ビット	**i, A**	**CLB 5, A**
ゼロページ・ビット	**i, zz**	**CLB 5, $04**
アキュムレータ・ビット相対	**i, A, hhll**	**BBC 5, A, *-12**
ゼロページ・ビット相対	**i, zz, hhll**	**BBC 5, $04, *-12**
ゼロページ即値	**#nn, zz**	**LDM #$77, $14**

命令ニモニック

ロード

命令	アドレス指定モード
LDA	即値
	ゼロページ
	ゼロページX
	絶対
	絶対X
	絶対Y
	ゼロページ間接X
	ゼロページ間接Y
LDM	ゼロページ即値
LDX	即値
	ゼロページ
	ゼロページY
	絶対
	絶対Y
LDY	即値
	ゼロページ
	ゼロページX
	絶対
	絶対X

格納

命令	アドレス指定モード
STA	ゼロページ
	ゼロページX
	絶対
	絶対X
	絶対Y
	ゼロページ間接X
	ゼロページ間接Y
STX	ゼロページ
	ゼロページY
	絶対
STY	ゼロページ
	ゼロページX
	絶対

転送

命令	アドレス指定モード
TAX	暗示
TXA	暗示
TAY	暗示
TYA	暗示
TSX	暗示
TXS	暗示

スタック操作

命令	アドレス指定モード
PHA	暗示
PHP	暗示
PLA	暗示
PLP	暗示

加算と減算

命令	アドレス指定モード
ADC,SBC	即値 ゼロページ ゼロページX 絶対 絶対X 絶対Y ゼロページ間接X ゼロページ間接Y
INC,DEC	アキュムレータ ゼロページ ゼロページX 絶対 絶対X
INX, DEX, INY, DEY	暗示

暗示乗算と除算

同じプロセッサでのみ利用可能です。

命令	アドレス指定モード
MUL, DIV	ゼロページX

論理演算

命令	アドレス指定モード
AND, ORA, EOR	即値
	ゼロページ
	ゼロページX
	絶対
	絶対X
	絶対Y
	ゼロページ間接X
	ゼロページ間接Y
COM, TST	ゼロページ
BIT	ゼロページ
	絶対

比較

命令	アドレス指定モード
CMP	即値
	ゼロページ
	ゼロページX
	絶対
	絶対X
	絶対Y
	ゼロページ間接X
	ゼロページ間接Y
CPX, CPY	即値
	ゼロページ
	絶対

シフトとローテート

命令	アドレス指定モード
ASL, LSR, ROL, ROR	アキュムレータ
	ゼロページ
	ゼロページX
	絶対
	絶対X
RRF	ゼロページ

ビット管理

命令	アドレス指定モード
CLB, SEB	アキュムレータ・ビット ゼロページ・ビット

フラグ設定

命令	アドレス指定モード
CLC, SEC, CLD, SED, CLI, SEI, CLT, SET, CLV	暗示

ジャンプ

命令	アドレス指定モード
JMP	絶対 間接絶対 ゼロページ間接
JSR	絶対 間接絶対 特殊ページ

分岐

命令	アドレス指定モード
BBC, BBS	アキュムレータ・ビット相対 ゼロページ・ビット相対
BCC, BLS, BNE, BEQ, BPL, BMJ, BVC, BVS	相対

リターン

命令	アドレス指定モード
RTI, RTS	暗示

割込み

命令	アドレス指定モード
BRK	暗示

その他

命令	アドレス指定モード
NOP	暗示

特殊

命令	アドレス指定モード
WIT, STP	暗示

第９章　XLINK リンカ

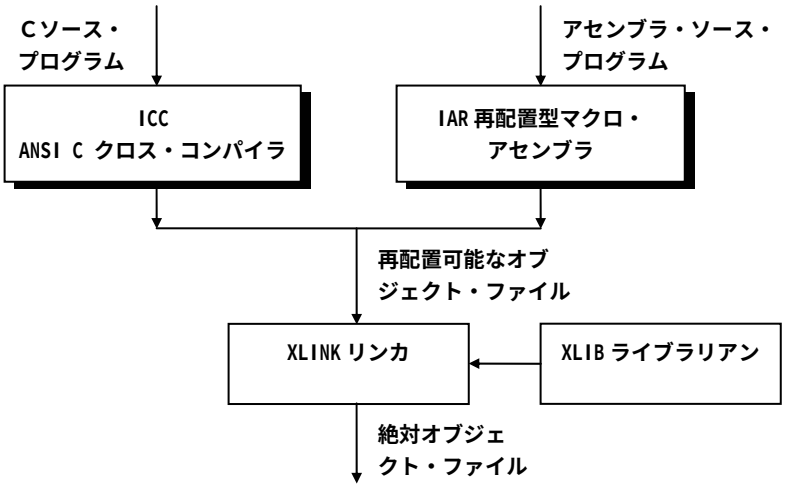
この章では IAR XLINK リンカを説明し、リンカの使用例を示します。

概要

XLINK リンカは、組み込みコントローラ・アプリケーションの開発での使用に適した強力で柔軟性のあるソフトウェア・ツールです。XLINK は IAR アセンブラやＣコンパイラが生成した１つ以上の再配置可能なオブジェクト・ファイルを読み取り、絶対機械コード・プログラムを出力として生成します。

XLINK リンカは、大きな再配置可能マルチモジュールのＣやアセンブラ・プログラムとの混合のリンクに適していると同時に、小さな単一ファイルの絶対アセンブラ・プログラムのリンクにも十分に適しています。

次の図は、リンク処理を示しています。



オブジェクト形式

IAR アセンブラやＣコンパイラで生成されるオブジェクト・ファイルは、UBROF と呼ばれる所有権を主張できる形式を使用します。UBROF とは、Universal Binary Relocatable Object Format のことです。アプリケーションは、任意の数の UBROF 再配置可能ファイルをアセンブラとＣのあらゆる組合せに構築することができます。

XLINK 機能

XLINK は、プログラムをリンクする場合に３種類の明確な機能を実行します。

- 実行可能なコードやデータをもったモジュールを入力ファイルからロードします。
- ユーザ指定アドレスでコードやデータの各セグメントを位置決めします。
- アセンブラやコンパイラが決定することができない全てのグローバル（すなわち、ローカルでない、プログラム全体に及ぶ）シンボルを決定することによって、各種のモジュールを相互にリンクします。
- プログラムで必要なモジュールをユーザ定義ライブラリからロードします。

ライブラリ

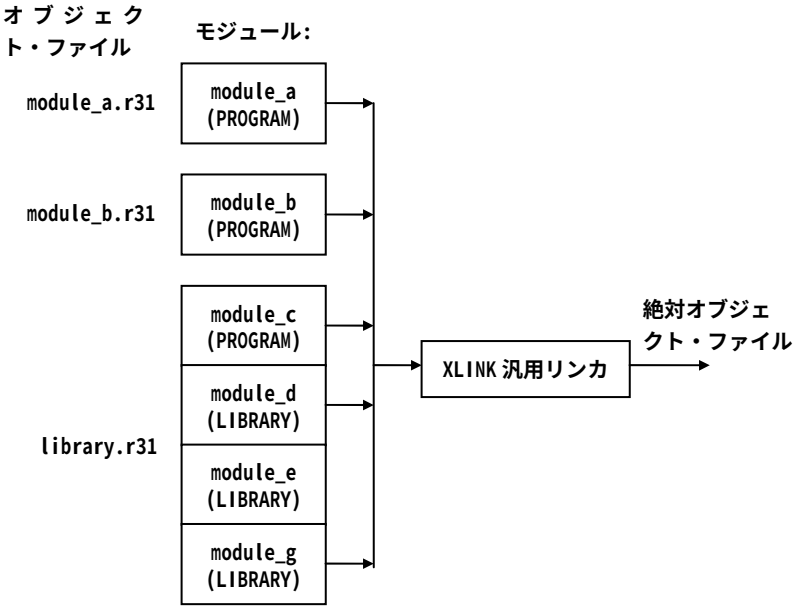
（複数のＣあるいはアセンブラ・モジュールをもつ）ライブラリ・ファイルを読み取ると、XLINK は、リンクするプログラムが実際に必要とするモジュールのみをロードします。これは、１ルーチンしか必要でないときに全てのモジュールをライブラリ・ファイルにロードしないようにするためです。これらのライブラリ・ファイルを管理するには、XLIB ライブラリアンを使用します。

出力形式

XLINK が生成する最終出力は、EPROM に入れたり、ハードウェア・エミュレータにダウンロードしたり、IAR C-SPY デバッガを使用して PC で実行したりすることができる絶対実行可能オブジェクト・ファイルです。

入力ファイルとモジュール

次の図は、XLINK が一般的なアセンブラ、Ｃプログラムの入力ファイル、ロード・モジュールを処理する方法を示しています。



メイン・プログラムは **module_a.s31** と **module_b.s31** の２つのソース・ファイルからアセンブルされ、２つの再配置可能なファイルを生成します。これらのファイルはそれぞれ、単一モジュール **module_a**、**module_b** から成ります。デフォルトでは、アセンブラは **module_a** と **module_b** の両方に PROGRAM 属性を割り当てます。これは、これらのモジュールが格納されているファイルが XLINK で処理されるたびにモジュールがロード、リンクされることを意味します。すなわち、ファイル名が引数として指定されます。

単一Ｃソース・ファイルからのコードやデータは、コンパイラで生成されるファイルの単一モジュールとなります。すなわち、Ｃソース・ファイルとＣモジュールの間には１対１の関係が成り立ちます。デフォルトでは、コンパイラはこのモジュールにオリジナルのＣソース・ファイルと同じ名前を与えます。複数のＣモジュールのライブラリは、XLIB を使用してのみ生成されます。

アセンブラ・プログラムは、１つのソース・ファイルに複数のモジュールがあるように構築することができます。各モジュールは、プログラム・モジュールでもライブラリ・モジュールでも構いません。

ライブラリ

上図では、**library.r31** ファイルは、それぞれがアセンブラやＣコンパイラによって生成される可能性をもった複数のモジュールから成ります。

PROGRAM 属性をもつモジュール **module_c** は、**library.r31** ファイルがリンカのために入力ファイルの中でリストされるたびにいつもロードされます。ランタイム・ライブラリでは、スタートアップ・モジュール **cstartup**（全てのＣプログラムで必要なモジュール）は、Ｃプロジェクトがリンクされるときに必ず組み込まれるように PROGRAM 属性をもっていま

す。

library.r32 ファイルのその他のモジュールは、LIBRARY 属性をもっています。ライブラリ・モジュールは、モジュールが、ロードされた別のモジュールによってなんらかの方法で参照されるエントリ（**PUBLIC** として宣言される関数、変数、あるいはその他のシンボル）をもっている場合にのみロードされます。XLINK は、このようにしてプログラム構成に必要なモジュールのみをライブラリ・ファイルから獲得し、それ以外は得ません。例えば、**module_e** のエントリが、ロードされたモジュールによって参照されない場合、**module_e** はロードされません。

これは、次のように機能します。

module_a が外部シンボルを参照すると、XLINK は残りの入力ファイルを検索し、**PUBLIC** エントリとしてそのシンボルをもつモジュール、つまりエントリ自体が配置されているモジュールの有無を調べます。XLINK は、**module_c** で **PUBLIC** として宣言されたシンボルを見つけ出すと、そのモジュールを（まだロードされていなければ）ロードします。この手続きは反復されますので、**module_c** が外部シンボルを参照すると、同じことが起こります。ライブラリ・ファイルはちょうどその他の再配置可能なオブジェクト・ファイルのようなものであると理解してください。実際、ライブラリという明確なファイルの型はありません（モジュールは、LIBRARY 属性や PROGRAM 属性をもっています）。ファイルをライブラリと見なすのは、ファイルが何を格納し、どのように使用されるかによって決まります。簡潔にいうと、ライブラリとは、要求があった場合にのみロードされるようにそのほとんどが LIBRARY 属性を備えた一連のよく使用される関連モジュールをもった **.r31** ファイルと言えます。

ライブラリの生成

Ｃ言語やアセンブラ・モジュールを使用して、自分のライブラリを生成したり、それを既存のライブラリに追加したりすることができます。Ｃモジュールにデフォルトの PROGRAM 属性ではなく LIBRARY 属性をもつようにさせるには、Ｃコンパイラの**-b** オプションを使用します。アセンブラ・プログラムでは、モジュールに LIBRARY 属性を指定する場合は**MODULE** 疑似命令を、また PROGRAM 属性を指定する場合は **NAME** 疑似命令を使用します。

ライブラリを生成、管理するには、XLIB ライブラリアンを使用します。数あるタスクの中で、XLIB ライブラリアンは、別のモジュールがコンパイルあるいはアセンブルされた後にそのモジュールの属性（PROGRAM/LIBRARY）を変更するために使用されます。

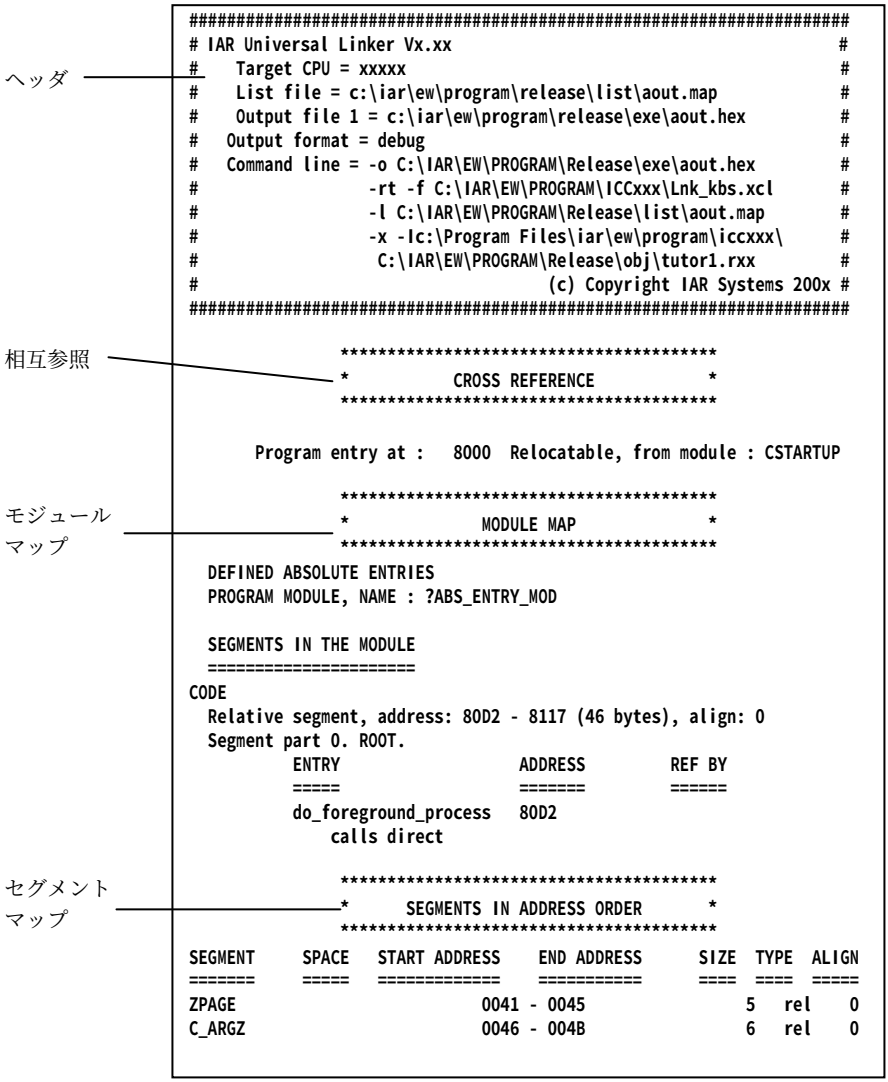
セグメント・ロケーション

XLINK がプログラムのためにロードするモジュールを識別すると、その最も重要な機能の１つは、プログラムで使用される各種のコードやデータ・セグメントにロード・アドレスを割り付けることです。

アセンブリ言語プログラムでは、プログラマは再配置可能なセグメントを宣言、命名し、その使用方法を決定する責任があります。Ｃプログラムでは、コンパイラが一連の定義済みコードやデータ・セグメントを生成、使用しますので、セグメントの命名や使用に関しては制約付きの制御しか行えません。

リストの形式

デフォルトの XLINK リスト形式は、以下に示すとおりです。

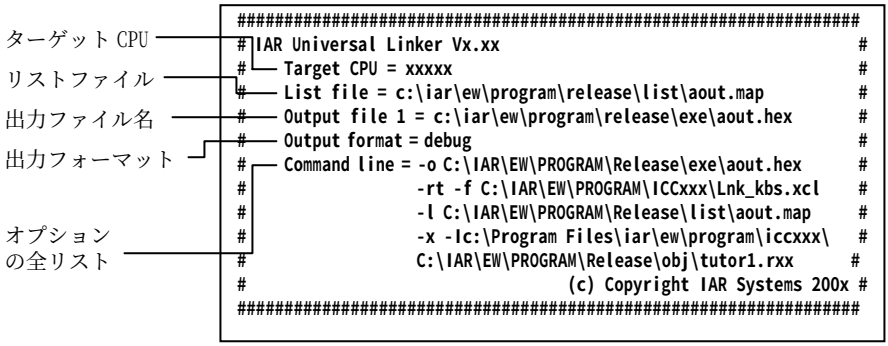



```
#############################################################################
# IAR Universal Linker Vx.xx                                                #
#    Target CPU = xxxxx                                                     #
#    List file = c:\iar\ew\program\release\list\aout.map                    #
#    Output file 1 = c:\iar\ew\program\release\exe\aout.hex                 #
#   Output format = debug                                                   #
#   Command line = -o C:\IAR\EW\PROGRAM\Release\exe\aout.hex                #
#                  -rt -f C:\IAR\EW\PROGRAM\ICCxxx\Lnk_kbs.xcl              #
#                  -l C:\IAR\EW\PROGRAM\Release\list\aout.map               #
#                  -x -Ic:\Program Files\iar\ew\program\iccxxx\             #
#                   C:\IAR\EW\PROGRAM\Release\obj\tutor1.rxx                #
#                                    (c) Copyright IAR Systems 200x #
#############################################################################

                  ****************************************
                  *           CROSS REFERENCE            *
                  ****************************************

       Program entry at :   8000  Relocatable, from module : CSTARTUP

                  ****************************************
                  *              MODULE MAP              *
                  ****************************************
  DEFINED ABSOLUTE ENTRIES
  PROGRAM MODULE, NAME : ?ABS_ENTRY_MOD

  SEGMENTS IN THE MODULE
  ======================
CODE
  Relative segment, address: 80D2 - 8117 (46 bytes), align: 0
  Segment part 0. ROOT.
          ENTRY                   ADDRESS         REF BY
          =====                   =======         ======
          do_foreground_process   80D2
              calls direct

                  ****************************************
                  *      SEGMENTS IN ADDRESS ORDER       *
                  ****************************************
SEGMENT     SPACE   START ADDRESS    END ADDRESS      SIZE  TYPE  ALIGN
=======     =====   =============    ===========      ====  ====  =====
ZPAGE                         0041 - 0045                5   rel    0
C_ARGZ                        0046 - 004B                6   rel    0
```

リストは、次のセクションから成ります。

ヘッダ

ヘッダは、コマンド行と、XLINK コマンド用に選択されたオプションを示します。



オプションの全リストは、コマンド行に指定されたオプションを示します。**-f** オプションで指定されたコマンド・ファイルのオプションもカッコで示されます。

相互参照

相互参照は、エントリ・リスト、モジュール・マップ、および／またはセグメント・マップから成ります。これは、ハードウェア・エミュレータのサポートのために一部の出力形式で使用されるプログラム・エントリ・ポイントを含んでいます。６８ページ「モジュール制御関数」のアセンブラ **END** 疑似命令を参照してください。

エントリ・リスト（-xe オプション）

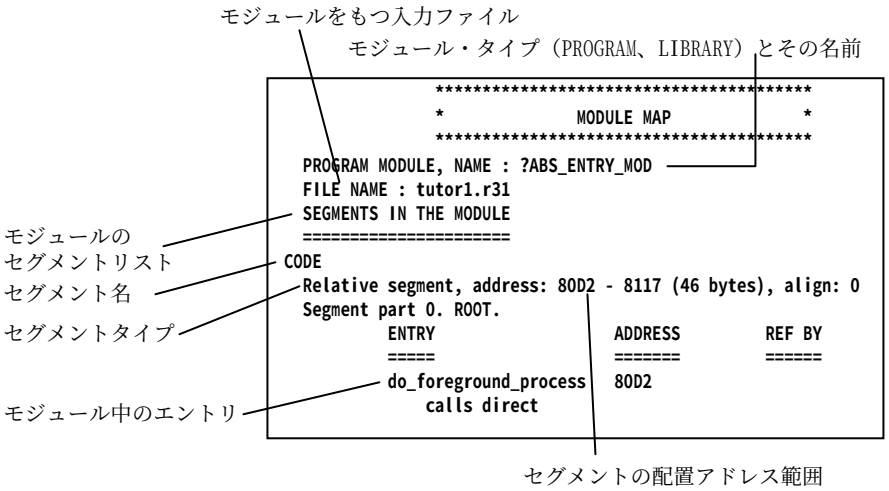
各モジュールとファイル名のエントリ名、アドレスを示します。

```
                ****************************************
                *              ENTRY LIST              *
                ****************************************
?ABS_ENTRY_MOD
tutor1 (tutor1.r31 )
  do_foreground_process                  80D2
  main                                   80E6
  con_char                               8123
  call_count                             004C
```

モジュール・マップ（-xm オプション）

モジュール・マップは、プログラムの一部としてロードされた各モジュールのサブセクションから成ります。各サブセクションは、次の情報を示します。

各セグメントについては、モジュール・マップはローカルとエントリもリストします。



セグメント・マップ（-xs オプション）

セグメント・マップは、アドレス順でセグメントをリストします。

```
**************************************
*      SEGMENTS IN ADDRESS ORDER     *
**************************************
SEGMENT     SPACE   START ADDRESS    END ADDRESS      SIZE  TYPE  ALIGN
=======     =====   =============    ===========      ====  ====  =====
ZPAGE                       0041 - 0045                 5   rel    0
C_ARGZ                      0046 - 004B                 6   rel    0
```

このマップは、各セグメントの開始アドレス、終了アドレス、および次のパラメータをリストします。

パラメータ	内容
TYPE	セグメントの型
	rel 相対
	stc スタック
	bnk バンクされた
	com 共通
ORG	起点、すなわちセグメント開始アドレス型
	stc 絶対。**ASEG** セグメント用
	flt 浮動。**RSEG**, **COMMON** あるいは **STACK** セグメント用
P/N	正／負、すなわちセグメントの割当方法
	pos 上向き。**ASEG**, **RSEG**, **COMMON** セグメント用
	neg 下向き。**STACK** セグメント用
ALIGN	セグメントは、次の **2^ALIGN** アドレス境界に合わせられる。

第１０章 XLINK オプション

XLINK オプションを使用すると、組み込みワークベンチまたは、コマンドラインから XLINK リンカの操作を制御することができます。

オプションは、組み込みワークベンチ版の XLINK オプションのページに対応して次の分類項目に分けられます。

Output　#define　Diagnostics　List　Include

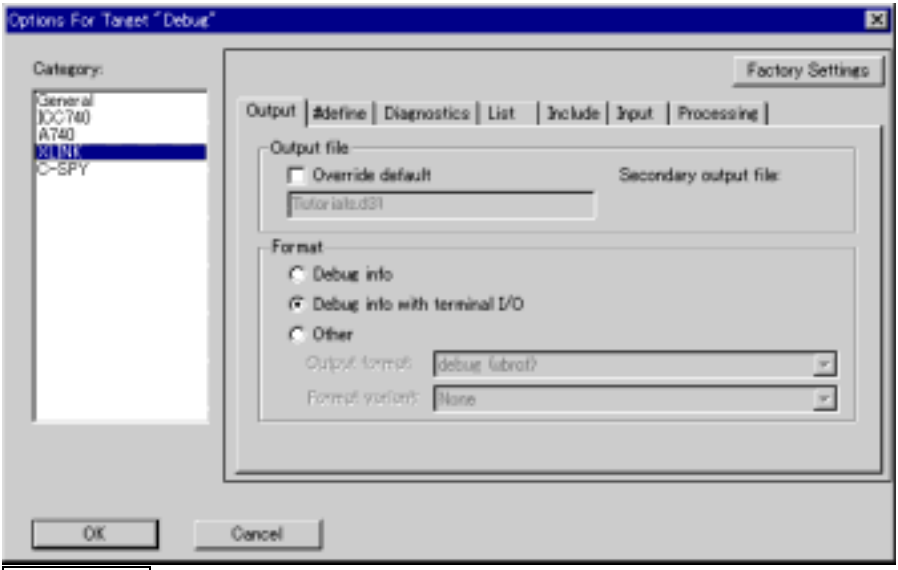
Input　Processing

Command line と Segment control の分類項目は、コマンドライン版または、XCL コマンド・ファイルでのみ利用できる追加的なオプションについての情報を提供しています。

XLINK オプションの設定

組み込みワークベンチ用 XLINK オプションの設定

組み込みワークベンチで XLINK オプションを設定するには、Project メニューから Options…を選んで、Category リストから XLINK を選択し、XLINK オプションのページを表示させます。



コマンドライン用 XLINK オプションの設定

コマンド行からオプションを設定するには、次のいずれかを行います。

◆　xlink コマンド後に、コマンド行で XLINK オプションを指定します。

- XLINK_ENVPAR 環境変数に XLINK オプションを指定します。
- XCL コマンド・ファイルに XLINK オプションを指定し、-f file コマンドでこのファイルをコマンド行に入れます。

XCL ファイル中には、C および C++スタイルの/* ... */ または、// コメントが使用できます。

オプションの概要

下表に、全ての XLINK オプションの概要を示します。各オプションの詳細説明については、次のセクションでオプションの分類名に従って参照してください。

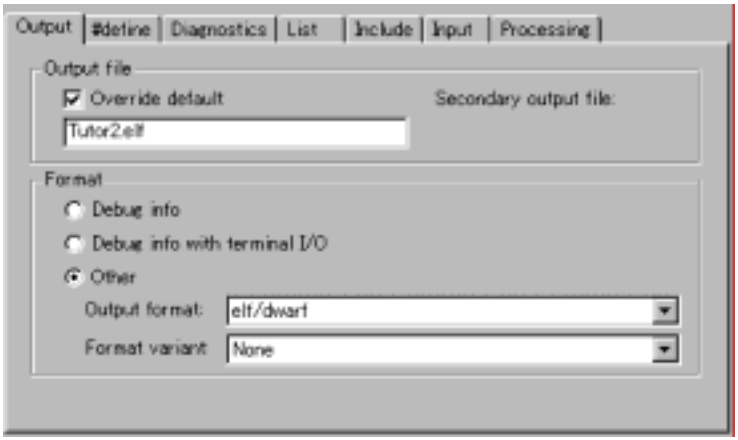
オプション	内容	分類項目
-! ｺﾒﾝﾄ -!	コメント・デリミタ。	Command line
-A ﾌｧｲﾙ, ...	入力ファイルを強制的にプログラム・モジュールとしてロードする。	Input
-a{i|w}[関数ﾘｽﾄ]	静的オーバーレイを不能にする。	Command line
-B	出力ファイルを強制的に生成する。	Diagnostics
-b ﾊﾞﾝｸ定義	バンク・セグメントを定義する。	Segment control
-C ﾌｧｲﾙ, ...	入力ファイルを条件付きでロードする。	Input
-c*cpu*	プロセッサ・タイプを指定する。	Command line
-D ｼﾝﾎﾞﾙ=値	シンボルを定義する。	#define
-d	コードの生成を不能にする。	Command line
-E ﾌｧｲﾙ,...	指定されたファイルにオブジェクト・コードを生成しないで、リンクを行う。	Input

オプション	内容	分類項目
-e 新=旧[,旧]...	外部シンボルをリネームする。	Command line
-F フォーマット	出力形式を選択する。	Output
-f ファイル	XCL ファイル名。	Include
-G	グローバル型チェックを不能にする。	Diagnostics
-H ヘキサ値	未使用コードメモリを充填する。	Processing
-I パス	インクルード・パス。	Include
-J サイズ,方法[,補数]	チェックサムを生成する。	Processing
-K*seg=inc,count*	重複コード。	Segment control
-L ディレクトリ	リスト・ファイルのディレクトリを指定する。	List
-l ファイル	リスト・ファイル名を指定する。	List
-n[c]	ローカル・シンボルを無視する。	Command line
-o ファイル	出力ファイル名。	Output
-P*pack_def*	パックされたセグメントを定義する。	Segment control
-p 行数	ページ毎の行数を設定する。	List
-R[w]	アドレス範囲のチェック。	Diagnostics
-r	デバッグ情報。	Output
-rt	端末入力出力つきデバッグ情報。	Output
-S	サイレント操作を選択する。	Command line
-w[番号|s|t]	警告メッセージを不能にする。	Diagnostics
-x[e][m][s]	相互参照リストを生成する。	List
-Y[文字列]	出力形式の可変部を選択する。	Output
-Z セグメント	セグメントを定義する。	Segment control
-z	セグメント・オーバーラップの エラーを警告にする。	Diagnostics

Output

Output オプションは、出力フォーマットとデバッグ情報のレベルを指定するのに用いられます。

組み込みワークベンチ用



コマンドライン用

-o ファイル	出力ファイル名
-r	Debug info
-rt	Debug info with terminal I/O
-F フォーマット	Output format
-Y[文字列]	Format variant
-y[文字列]	Format variant

出力ファイル名（-o）

構文：　-o ファイル

XLINK 出力ファイル名を指定するには、Output file(-o)を使用します。ファイル名が指定されないと、リンカはファイル名 aout.hex を使用します。ファイル・タイプ(拡張子)を指定しないでファイル名を与えた場合は、選択された出力形式 (Output format(-F)オプション) のデフォルトのファイル・タイプが使用されます。

出力ファイルを2個生成する出力形式を選択した場合、ユーザ指定ファイル・タイプは主要出力ファイル (第一のフォーマット) に影響を与えるのみです。

デバッグ情報（-r）

構文：　　-r

DEBUG (UBROF)形式のファイルを拡張子.d31 をつけて出力し、SW シミュレータ、あるいは UBROF 形式をサポートするエミュレータで使用するには、Debug info(-r)を使用します。

Debug info (-r)の指定は、いかなる Output format (-F)オプションをも上書きします。

端末 I/O シミュレーション付きデバッグ情報（-rt）

構文：　　-rt

SW デバッガ付きの出力ファイルを使い、端末 I/O をエミュレートするには、Debug info with terminal I/O (-rt)を用います。

出力形式を選択する（-F）

構文：　　-F フォーマット

出力形式を選択するには、Output format(-F)を使用します。

デフォルトの出力形式をシステムにインストールするには、環境変数 XLINK_FORMAT を設定します。

サポートされた XLINK 出力形式の1つ。出力形式の詳細については、「XLINK 出力形式」を参照してください。

出力形式が指定されない場合は、デフォルトの INTEL-STANDARD が使用されます。

Output format(-F)オプションを DEBUG にしても SW デバッグがサポートされないことに注意してください。かわりに、Debug info(-r)オプションを用いてください。

出力形式の可変部を選択する（-Y）

構文：　　-Y[文字列]

一部の出力形式で利用できる拡張機能を選択するには、Format variant(-Y)を使用します。詳細については、「XLINK 出力形式」を参照してください。

Format variant におけるオプションは、選択した出力形式に依存します。

出力形式の可変部を選択する（-y）

構文：　　-y[文字列]

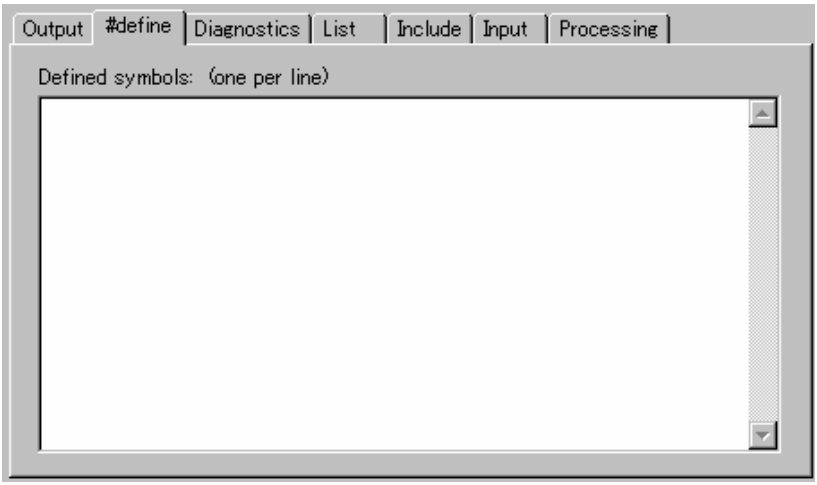
一部の出力形式で利用できる拡張機能を選択するには、Format variant(-y)を使用します。新オプション-y にのあとに一連の文字列が指定できます。影響されるフォーマットは、IEEE695、XCOFF78K および ELF です。

詳細については、「XLINK 出力形式」の章を参照してください。

#define

#define オプションは、シンボルの定義を可能にします。

組み込みワークベンチ用



コマンドライン用

-D シンボル=値　　シンボルを定義する。

シンボルを定義する（-D)

構文：　-D シンボル=値

ここで、シンボルは、他の部分では定義されていない、プログラムの中の外部 (EXTERN) シンボルで、「値」は、シンボルに割り付ける値。

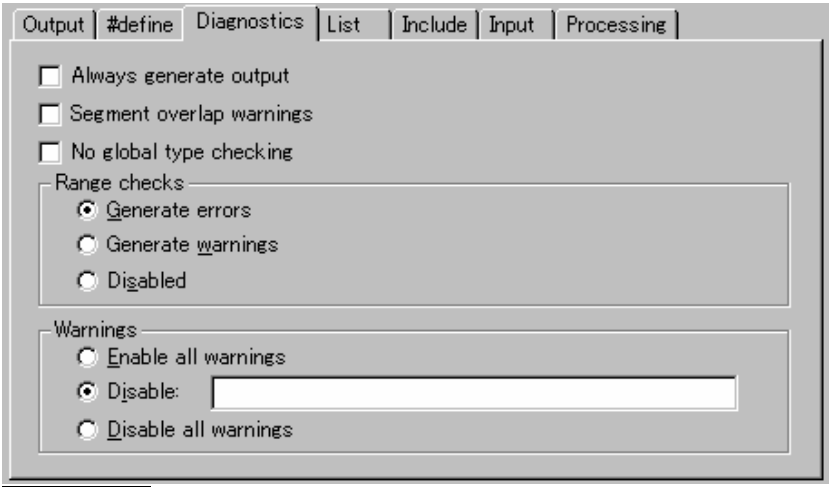
リンク時に絶対シンボルを定義するには、Define symbol(-D)を使用します。これは、コンフィギュレーションの目的には特に有用です。シンボルは、XLINK 操作の XCL ファイル・モードを使用すると、いくらでも定義できます。このようにして定義されたシンボルは、?ABS_ENTRY_MOD と言うリンカによって生成される特殊モジュールに属します。

既存のシンボルを再定義しようとすると、XLINK はエラー・メッセージを表示します。

Diagnostics

Diagnostics オプションは、XLINK リンカが生成するエラーと警告メッセージを決定します。

組み込みワークベンチ用



コマンドライン用

-B	Always generate output
-R[w]	Disable range check
-w[番号|s|t]	Disable warnings
-z	Segment overlap warnings
-G	No global type checking

強制的に出力ファイルを生成する（-B)

構文： -B

グローバル・エントリが、ないまたは多重に定義されているなどの、致命的でないエラーがリンクの過程で発生した際でも出力ファイルを生成するには、Always generate output(-B)を用います。通常 XLINK は、エラーを検知すると、出力ファイルの生成を行いません。たとえ -B を指定しても、致命的エラーが発生すれば、XLINK は処理をやめることに注意してください。

Always generate output (-B)オプションは、絶対出力イメージ中で、欠けているエントリをあとでパッチすることを可能にします。

セグメント・オーバーラップエラーを警告にする（-z)

構文： -z

セグメントのオーバーラップ・エラーを警告に軽減させ、相互参照マップなどが生成されるようにする

には、Segment overlap warnings(-z)を使用します。

グローバル型チェックを不能にする（-G)

構文：　　-G
リンク時に型チェックを不能にするには、No global type checking(-G)を使用します。適切に記述されたプログラムはこのオプションを必要としないはずですが、このオプションが有用になるような場合もあります。

省略時には、XLINK は外部参照を PUBLIC エントリをもつエントリと比較して、モジュール間でリンク時の型チェックを実行します（関連のオブジェクト・モジュールに情報が存在する場合に行われます）。モジュール間で不一致があると、警告が出力されます。

アドレス範囲のチェックを不能にする（-R)

構文：　　-R[w]
アドレス範囲のチェックをするには、Range checks(-R)を使用します。

アドレスがターゲット CPU のアドレス範囲（コード、外部データ、あるいは内部データ・アドレス）を超えて再配置された場合は、エラー・メッセージが発生します。通常これは、アセンブリ言語モジュールやセグメント配置におけるエラーを指します。

組み込みワークベンチ用

アドレス範囲のチェックをするには、Range checks を使用します。以下の表は、組み込みワークベンチにおけるアドレス範囲チェックオプションを示しています。

EW オプション	説明
Generate errors	エラーメッセージを発生します。
Generate warnings	アドレス範囲エラーを警告として扱います。
Disabled	アドレス範囲チェックを不能にします。

コマンドライン用

アドレス範囲チェックを不能にするには、-R を使用します。XLINK がアドレス範囲エラーを警告として扱うようにさせるには、-Rw を使用します。

警告メッセージを不能にする（-w)

構文：　　-w[警告|s|t]
警告メッセージを抑制するには、Disable warnings(-w)を使用します。

オプションの修飾子は、どの警告を不能にするかを指定します。たとえば、警告 3 と 7 を不能にするには、以下のように指定します。

-w3 -w7

-ws を指定すると、XLINK の返値が次のように変わります。

条件	デフォルト	-ws
エラーまたは警告なし	0	0
警告はあるがエラーはなし	0	1
1 つ以上のエラー	2	2

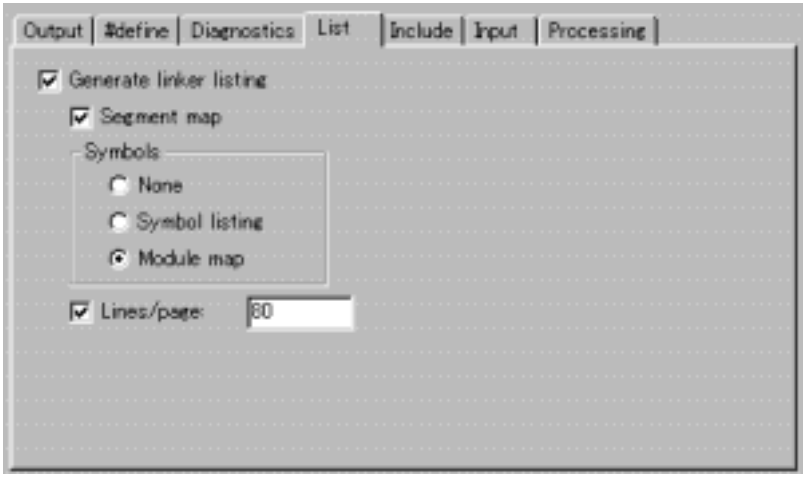
警告 6 (型不一致) および 35 (同一のタグに対して複数の struct) で与えられる詳細な型情報を抑制するには、-wt を使用します。

修飾子が省略された場合、すべての警告が不能にされます。

List

List オプションは、XLINK の相互参照リストの生成を決定します。

組み込みワークベンチ用



コマンドライン用

-l ﾌｧｲﾙ　　リスト･ファイル名を指定する。

-L ﾃﾞｨﾚｸﾄﾘ　　リスト･ファイルのディレクトリを指定する。

-x[e][m][s]　　相互参照リストを生成する。

-p 行数　　ページ毎の行数を設定する。

組み込みワークベンチ用

Generate linker listing

リンカのリストを *project*.map ファイルに生成します。

コマンドライン用

リスト・ファイル名を指定する (-l)

構文： -l ﾌｧｲﾙ

リンカ･リストを指定したファイルに生成します。拡張子を指定しない場合は、デフォルトの.lstが使用されます。しかしながら、リンカのリスト･ファイルをアセンブラやコンパイラのリスト･ファイルと混同しないようにするために、拡張子.mapの使用を勧めます。

-lは-Lと同時に使用できません。

リスト・ファイルのディレクトリを指定する（ーL)

構文： -L ﾃﾞｨﾚｸﾄﾘ

リンカがリストを生成し、これを ﾃﾞｨﾚｸﾄﾘ\出力ﾌｧｲﾙ.lst ファイルに送るようにします。接頭辞の前にスペースを指定できないことに注意してください。

デフォルトでは、リンカはリストを生成しません。単純にリストを生成するには、接頭辞なしで-Lオプションを使用します。リストは拡張子.lstを除いてソースと同じ名前になります。

-Lは-lと同時に使用できません。

相互参照リストを生成する (-x)

構文： -x[e][m][s]

コマンドライン用

XLINKリスト･ファイルにセグメント･マップを含ませるには、Cross reference(-x)を使用します。

組み込みワークベンチ、コマンドライン

次のオプションが指定できます。

EW オプション	コマンドライン	説明
Segment map	s	ダンプの順序に並べられた全セグメントのリスト
Symbol listing	e	各モジュールの中の全てのエントリ（グローバル・シンボル）の省略形リスト。このエントリ・マップは、ルーチンやデータ・エレメントのアドレスを素早く見つけ出す場合に有用です。
Module map	m	プログラムの各モジュールの全てのセグメント、ローカル・シンボル、エントリ（公開シンボル）のリスト。

コマンドライン用

任意指定のパラメータを設定せずに-x を指定した場合は、デフォルトの相互参照リストが生成されます。これは、 -xms に相当します。この相互参照リストには、次のものが含まれます。

- 基本プログラム情報のあるヘッダー部
- シンボル相互参照付きのモジュール・ロード・マップ
- ダンプの順序のセグメント・ロード・マップ

相互参照リスト情報は、-l や-L オプションを指定しない場合には、スクリーンに出力されます。

ページ毎の行数を設定する（-p）

構文： -p 行数

XLINK リストのページ長さを「行数」に設定するには、-p を使用します。この値は、10～150 の範囲になければなりません。

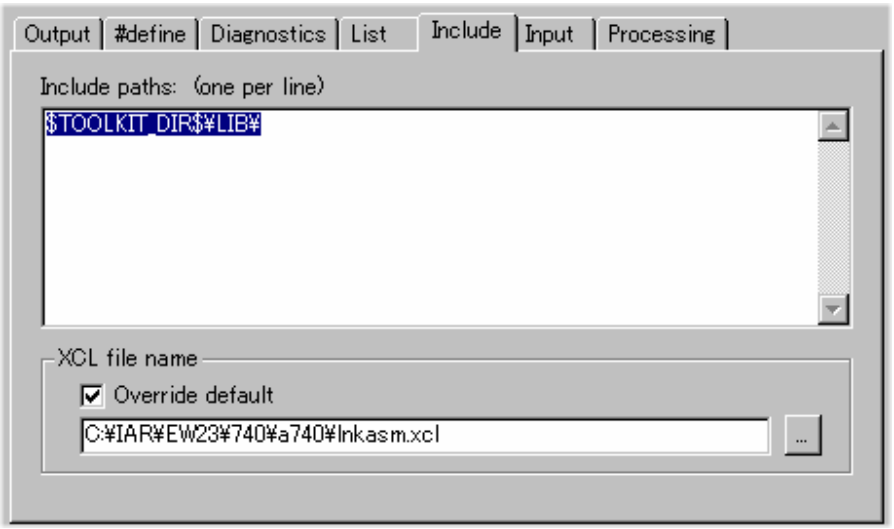
コマンドライン用

デフォルトのページ長さをシステムにインストールするには、環境変数 XLINK_PAGE で設定できます。

Include

Include オプションは、リンク・コマンド・ファイルに対するインクルード・パスの設定と、リンカ・コマンド・ファイルの指定を可能にします。

組み込みワークベンチ用



-I パス	インクルード・パス
-f ファイル	XCL ファイル名

インクルード・パス（-I）

構文： -I パス

オブジェクト・ファイルが探索されるパス名を指定します。

デフォルトでは、XLINK はオブジェクト・ファイルを現在のワーク・ディレクトリ内でのみ探索します。Include paths(-I)オプションは、現在のワーク・ディレクトリ内にこのファイルが見つからなったときに探索するディレクトリを指定することを可能にします。

これは、XLINK_DFLTDIR 環境変数と同等です。

XCL ファイル名（-f）

構文：　　-f ファイル

コマンドライン用

オプションをコマンドラインに入力したかのようにしてコマンド・ファイルから読み取って XLINK コマンド行を拡張するには、-f を使用します。

引数は、コマンドラインの場合と同じ構文を使用してテキスト・エディタで XCL ファイルに入力します。しかし、引数間の有効なデリミタについては、空白やタブに加えて行の終了 CR もデリミタとして扱います。コマンドラインは、\ ＋ ＜改行＞ シーケンスで拡張することができます。

C および C++スタイルのコメント/* … */ または // を、XCL ファイル内で指定することが可能です。

組み込みワークベンチ用

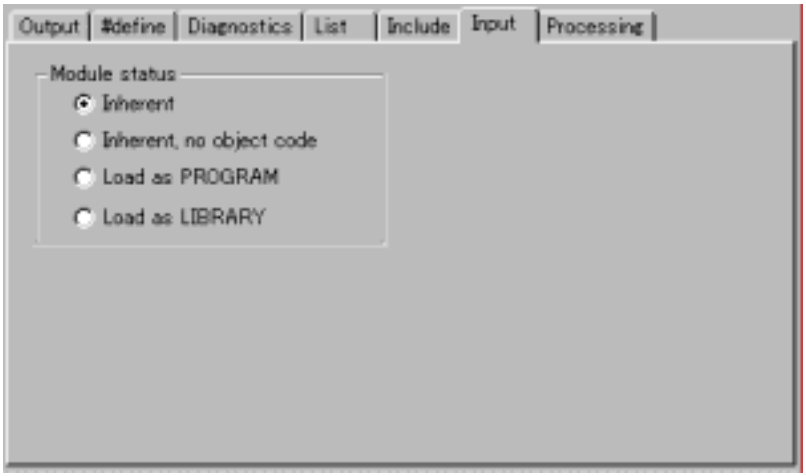
デフォルトの XCL ファイルは選択された General Target メモリ・モデルとプロセッサ・コンフィグレーションに対して、自動的に選ばれます。デフォルトのものに上書きするには、Override default を選び、代わりのコマンド・ファイルを選択します。

C および C++スタイルのコメント/* … */ または // を、XCL ファイル内で指定することが可能です。

Input

Input オプションは、入力モジュールの状態を決定します。

組み込みワークベンチ用



コマンドライン用

ファイル,... 普通のリンク。

-E ファイル,... 指定されたファイルにオブジェクト・コードを生成しないで、リンクを行う。

-A ファイル, ... 入力ファイルを強制的にプログラム・モジュールとしてロードする。

-C ファイル, ... 入力ファイルを条件付きでロードする。

組み込みワークベンチ用

Inherent

構文： ファイル,...

普通にファイルをリンクして出力コードを生成するには、Inherent を使用します。

指定されたファイルにオブジェクト・コードを生成しない（-E）

構文：　　-E ファイル,...

指定された入力ファイルを空ロードするには、Inherent, no object code(-E)を使用します。入力ファイルはすべての点で通常のリンクで処理されますが、出力コードはこれらのファイルに対して生成されません。

この機能の潜在的な使用法の1つは、複数の EPROM をプログラムするための個々の出力ファイルを生成することです。これは、出力ファイルに入れたいものを除いてすべての入力ファイルを空ロードすることによって行われます。

コマンドライン用

次の例では、プロジェクトは file1 から file4 の4つのファイルから構成されるが、file4 のオブジェクト･コードのみを生成し、EPROM に入れたいとします。

```
-E file1,file2,file3
file4
-o project.hex
```

v:\general\lib および c:\project\lib からオブジェクト･ファイルを読み取るには、次のようにします。

```
-Iv:\general\lib;c:\project\lib
```

入力ファイルをプログラム・モジュールとしてロードする（-A）

構文：　　-A ファイル, ...

たとえ一部のモジュールが LIBRARY 属性をもっていても、強制的に指定入力ファイル内の全てのモジュールをプログラム･モジュールであるかのように一時的にロードさせるには、Load as PROGRAM(-A)を使用します。

このオプションは、特にライブラリ･モジュールをライブラリ･ファイルにインストールする前にライブラリ･モジュールを検査するのに適しています。それは、-A オプションは同じエントリをもつ既存のライブラリ･モジュールを無効にするからです。すなわち、XLINK は、ライブラリ･モジュールで同じ名前のエントリをもつモジュールをではなく、-A 引数で指定された入力ファイルからのモジュールをロードします。

たとえば CLIB ライブラリの標準モジュールではなく、ユーザ記述ライブラリ･モジュール putchar.r31 をロードするには、次のようにします。

```
-! these lines are in an XCL file ... -!
-A putchar
CLIB
```

これは、putchar ファイルが CLIB の中のモジュールの1つと同じグローバル･エントリをもつものと想定されます。

入力ファイルをライブラリとしてロードする（-C)

構文：　　-C ﾌｧｲﾙ, ...

たとえ一部のモジュールが PROGRAM 属性をもっていても、強制的に指定入力ファイル内の全てのモジュールをライブラリ･モジュールであるかのように一時的にロードさせるには、Load as LIBRARY(-C)を使用します。これは、入力ファイルのモジュールが別のロード･モジュールで参照されるエントリをもつ場合にのみロードされることを意味しています。

たとえば CLIB の中の同じ名前のプログラム･モジュールではなく cstartup.r31 ファイルからのユーザ定義 CSTARTUP モジュールをロードするには、次のようにします。

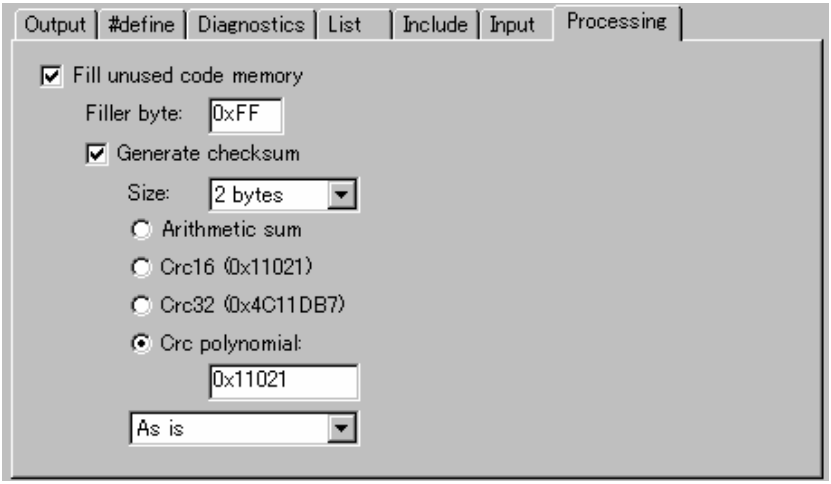
```
-! these lines are in an XCL file ... -!
cstartup
-C CLIB
```

これによって、CSTARTUP モジュールをライブラリにインストールする前にこのモジュールを検査することができます。

Processing

Processing オプションは、コードがどのように生成されるかを決定する追加的なオプションを指定します。

組み込みワークベンチ用



コマンドライン用

-H ﾍｷｻ値　　未使用コードメモリを充填する。

-J ｻｲｽﾞ,方法[,補数]　　チェックサムを生成する。

Fill unused code memory (-H)

構文：　　-H ﾍｷｻ値

リンカで生み出されたセグメント部分の間ギャップをﾍｷｻ値の値で満たすには、Fill unused code memory (-H)を使用します。リンカは並べ替えの制限または、セグメント配置オプションで与えられた領域の終端において、このギャップを生み出すことが起こり得ます。-H オプションがない、通常の場合では、出力ファイル中にこれらのギャップに値が与えられません。

例

-HFF

はすべてのギャップを 0xFF で満たします。

Generate checksum (-J)

構文：　　-J ｻｲｽﾞ,方法[,補数]

生成された生のデータすべてのチェックサムをとるには、Generate checksum (-J) を使用します。こ

のオプションは、Fill unused code memory (-H)オプションが指定された場合のみ指定可能です。

ｻｲｽﾞはチェックサムのバイト数を指定し、1、2、または 4 のどれかの値をとります。

「方法」は使用されるアルゴリズムを指定し、以下の内の 1 つの値をとります。

方法	説明
sum	単純な算術加算
crc16	CRC16 (多項式 0x11021 を生成する)
crc32	CRC32 (多項式 0x4C11DB7 を生成する)
crc=*n*	*n* の多項式を生成する

「補数」は、1 の補数に対しては 1、または 2 の補数に対しては 2 の値をとります。

すべての場合これは、プロセッサの自然なバイト順で、出力される結果の最下位の 1、2 または 4 バイトです。CRC チェックサムは、あたかも続くコードが入力においてそれぞれのビットに対してコールされ、0 バイトの引数をとり、0 の CRC で開始されます。

```
unsigned long
crc(int bit, unsigned long oldcrc)
{
        unsigned long newcrc = (oldcrc << 1) ^ bit;
        if (oldcrc & 0x80000000)
                newcrc ^= POLY;
        return newcrc;
}
```

POLY は生成多項式です。チェックサムは、この最後のルーチンコールの結果です。もし補数が与えられれば、チェックサムはこの結果の 1 または 2 補数になります。

リンカはチェックサム・バイトを CHECKSUM セグメントのラベル__checksum におきます。このセグメントは、他のセグメントと同様にセグメント配置オプションを用いて、配置される必要があります。

例

```
-J4,crc32,1
```

これは、生成多項式 0x4C11DB7 を用いて 4 バイトのチェックサムを算出し、計算値の 1 の補数を出力します。

コマンドライン

次の追加的なオプションは、コマンドラインから、または XCL ファイル内でのみ設定可能です。

-! ｺﾒﾝﾄ -!	コメント・デリミタ。
-a{i|w}[関数ﾘｽﾄ]	静的オーバーレイを不能にする。
-c740	プロセッサ・タイプを指定する。
-d	コードの生成を不能にする。
-e 新=旧[,旧]...	外部シンボルをリネームする。
-n[c]	ローカル・シンボルを無視する。
-S	サイレント操作を選択する。

コマンドライン用 コメント・デリミタ (-!)

構文： -! ｺﾒﾝﾄ -!

XLINK の.xcl ファイルでコメントを囲むには、-!を使用します。-!は、行の先頭から記述されていなければ、その直前に空白やタブをいれてください。

ファイル中には、C スタイルまたは C++スタイルのコメントも使用できます。これらは-!よりエラーを発生させる危険性を低減させますので、推奨されます。

コマンドライン用 静的オーバーレイを不能にする(-a)

構文： -a{i|w}[関数ﾘｽﾄ]

変数の静的メモリ配置を制御するには、-a を使用します。オプションは以下のとおりです。

オプション	説明
-a	デバッグのために、オーバレイを総合的に不能にします。
-ai	間接ツリー・オーバーレイを不能にします。
-aw	警告 16、Function is called from two function trees を不能にします。コードが正しいと確信できるときのみこれを使用してください。

さらに-a オプションは、指定された関数についての追加的なオプションを指定するために、1 つ以上の関数を指定可能です。それぞれの関数リストは、以下の形式をもちます。ここで function は、PUBLIC 関数または module:function の結合です。

関数リスト	説明
(function,function ...)	関数ツリーは他の関数とオーバーレイしない。
[function,function ...]	関数ツリーは、もし他の関数からコールされないならば、配置されない。
{function,function ...}	指定された関数が、割り込み関数。

複数の-a オプションが指定でき、それぞれの-a オプションもまた、複数のサブ・オプションをふくむことが可能です。

コマンドライン用 プロセッサ・タイプを指定する (-c)

構文： -c740

CPU タイプを 740 に設定するには、-c を使用します。

-c オプションの省略値をコマンド行で指定する必要がないようにするためにこのデフォルト値をインストールするには、環境変数 XLINK_CPU を使用します。

コマンドライン用 コードの生成を不能にする (-d)

構文： -d

XLINK からの出力コードの生成を不能にするには、-d を使用します。このオプションは、プログラムの試験リンクに有用です。例えば、構文エラーやシンボル定義の抜けなどのチェックです。このオプションを使用すると、大容量のプログラムの場合、XLINK は多少高速に実行されます。

コマンドライン用 外部シンボルをリネームする (-e)

構文： -e 新=旧[,旧]...

既存のシンボルを再定義しようとすると、XLINK はエラー・メッセージを表示します。

関数呼出しをある関数から別の関数へ向けて定義し、リンク時にプログラムを構成するには、-e を使用します。

これは、スタブ関数を生成する場合にも使用されます。すなわち、システムがまだ完了していない場合、未定義の関数呼出しは実際の関数が記述されるまでダミー・ルーチンに向けられます。

コマンドライン用 ローカル・シンボルを無視する (-n)

構文： -n[c]

入力モジュールのローカル（非公開）シンボルを全て無視するには、-n を使用します。

コンパイラが生成したローカル・シンボルのみを無視するには、-nc を使用します。

このオプションはリンク処理のスピードアップを図り、リンク完了に必要なホスト・システムのメモリ容量を削減することもできます。-n を使用すると、ローカル・シンボルは相互参照リストには示されず、出力ファイルにも渡されません。

ローカルシンボルは、適切なオプションでローカル・シンボルの組込みを指定してコンパイルやアセンブルを実行した場合にのみ、ファイルに組み込まれます。

コマンドライン用 サイレント操作を選択する (-S)

構文： -S

XLINK サインオン・メッセージや最終統計値レポートをオフして、リンカの実行中には画面に何も表示されないようにするには、-S を使用します。しかしながら、このオプションはエラーや警告メッセージ、あるいはリスト出力を不能にはしません。

セグメント制御

これらのオプションは、セグメントの配置を制御します。

-K*seg=inc,count*　　重複コード。

-P*pack_def*　　パックされたセグメントを定義する。

-Z ｾｸﾞﾒﾝﾄ　　セグメントを定義する。

-b ﾊﾞﾝｸ定義　　バンク･セグメントを定義する。

(-Z)と(-P)を使用するセグメント配置は、直前の配置を考慮に入れて一度に 1 つの配置を実行します。それぞれの配置が実行されるにつれて、すでの使用されている配置コマンドに対し与えられた範囲の任意の部分が、考慮された範囲から除去されます。メモリ範囲は、前のセグメント配置コマンドで配置されたセグメント、セグメント重複、または入力フィールドの絶対アドレスに配置されたオブジェクトによって、使用される可能性があります。

たとえば、ゼロページ（0～FF)に置かれるべき 2 つのセグメント(Z1，Z2)、および使用可能な RAM の任意の位置に置かれる 3 つのセグメント(A1，A2．A3)がある場合、これらは以下のように配置できます。

```
-Z(DATA)Z1,Z2=0-FF
-Z(DATA)A1,A2,A3=0-1FFF
```

これにより、Z1 と Z2 は 0 から上へ置かれ、これらが与えられた範囲に合わない場合にはエラーが発生し、そして A1、A2、および A3 は Z1 と Z2 で使用されていない最初のアドレスから配置されます。

(-P)オプションは、セグメント（またはセグメント部分）を連続して配置する必要がないという点で、(-Z)オプションと異なっています。(-P)を用いると、セグメント部分を前の配置で残されたホールに置くことが可能になります。

セグメントを 1 つの連続したかたまりにしておく必要がある場合、またはセグメント内のセグメント部分の順序を保存する必要がある場合、またはセグメントを指定した順に配置する必要がある場合には、(-Z)オプションを使用します。このようにするにはいくつかの理由がありますが、しかし大抵の理由は、かなりあいまいです。これを行う上でもっとも重要なことは、変数とそれらの初期化用データを 1 つのブロックに同じ順で配置することです。UBROF 7 以降を用いるコンパイラは、リンカが正しい振る舞いを行うように司令する属性を出力しますので、これらのコンパイラについてはもはや、変数初期化用データに対する(-Z)は必要とされません。

たとえばバンクのように、セグメントをいくつかの範囲に置く必要がある場合には、(-P)を使用します。

可能な場合には、(-b)オプションの代わりに(-P)を使用します。なぜならば、一般的に(-P)のほうがより強力で便利だからです。(-b)オプションは主に、後方互換の理由のためにサポートされますが、(-P)オプションを使ったときにはサポートされない場合も残っています。

ビットセグメントは、この配置コマンドが与えられているかどうかに関係なく、つねに最初に置かれます。

コマンドライン用 重複コード(-K)

構文： -K*seg*=*inc*,*count*

任意の生データバイトを *seg* にあるセグメントから *count* 回重複させ、毎回アドレスに *inc* を加算するには、(-K)を使用します。これは(-Z)オプションで説明されたセグメントに対して、典型的に使用されます。

このオプションは、PROM 全体が物理的にバンクになっているが、非バンクの PROM の部分を作成する場合に使用することができます。バンク・セグメントを残りの PROM に置くには、(-b)または(-P)オプションを使用します。

たとえば、RCODE0 と RCODE1 セグメントを、アドレス 0x20000 以降に 4 回重複させるには、以下のように指定します。

```
-KRCODE0,RCODE1=20000,4
```

これは、アドレス x、x+0x20000、x+0x40000、x+0x60000、および x+0x80000 にセグメントよりのバイトのそれぞれのコピーを 5 個出力ファイルに配置します。

コマンドライン用 パック・セグメントを定義する (-P)

構文： -P [(type)] segments=range[,range] ...

ここで、パラメータは以下のとおりです。

type　ターゲット・プロセッサに適用可能な場合は、すべてのセグメントにメモリ・タイプを指定します。このパラメータを省略すると、デフォルトの UNTYPED が使用されます。

segments　カンマで区切られた、リンクする1つ以上のセグメントのリスト。

range　start-end　start から始まって end で終了する範囲。

[start-end]*count+offset　count 範囲を指定します。ここで、最初は start から end までで、次は start+offset から end+offset のようになります。+offset 部分はオプションで、デフォルトは範囲の長さになります。

[start-end]/pagesize

pagesize のサイズと整合のページで分割された、start から end までの全領域を指定します。(注意) 範囲の start と end は、ページ境界と一致してはなりません。

以下の例は、アドレス範囲書式を示しています。

0-9,100-1FF	2 つの範囲。1 つは 0 から 9F まで、1 つは 100 から 1FF まで
[1000-1FFF]*3+2000	3 つの範囲。 1000-1FFF, 3000-3FFF, 5000-5FFF
[1000-1FFF]*3	3 つの範囲。 1000-1FFF, 2000-2FFF, 3000-3FFF
[50-77F]/200	5 つの範囲。 50-1FF, 200-3FF, 400-5FF, 600-77F

セグメント配置コマンドライン・オプションのすべての数値は、先頭に . (ピリオド) がつけられていない場合には、16 進と解釈されます。すなわち、10 および .16 と記述された数値は、ともに十進で 16 と解釈されます。

セグメント部分を、指定されたセグメントから指定された範囲にパッキングするには、(-P)を使用します。ここで、セグメント部分は、1 つのモジュールからのセグメントの一部として定義されます。リンカはそれぞれのセグメントをそれぞれの部分に分けて、新たなセグメントを領域ごとに形成します。すべての領域は閉じて (開始と終了の両方がある) いなければなりません。セグメント部分は、指定された順で領域に置かれることはありません。

コマンドライン用 セグメントを定義する (-Z)

構文： -Z [(type)] segments [=|#]range[,range] ...

ここで、パラメータは以下のとおりです。

type　ターゲット・プロセッサに適用可能な場合は、すべてのセグメントにメモリ・タイプを指定します。このパラメータを省略すると、デフォルトの UNTYPED が使用されます。

segments　カンマで区切られた、リンクする1つ以上のセグメントのリスト

セグメントは、リストと同じ順番でメモリに割り付けられます。セグメント名に +nnnn を付加すると、XLINK はそのセグメントに割り付けるメモリ容量を nnnn バイトだけ増加させます。

= or #　　セグメントの割付方法を指定します。

＝　　セグメントが指定範囲の先頭から始まるようにセグメントを割り付けます（上向き割付け）。

＃　　セグメントが指定範囲の終りで終了するようにセグメントを割り付けます（下向き割付け）。

割付演算子（およびアドレス範囲）を指定していない場合、セグメントはリンクされた最後のセグメントから上向きに割り付けられます。セグメントが全くリンクされていない場合は、アドレス0から割り付けられます。

range　　start-end　　start から始まって end で終了する範囲。

[start-end]*count+offset

count 範囲を指定します。ここで、最初は start から end までで、次は start+offset から end+offset のようになります。+offset 部分はオプションで、デフォルトは範囲の長さになります。

[start-end]/pagesize

pagesize のサイズと整合のページで分割された、start から end までの全領域を指定します。（注意）範囲の start と end は、ページ境界と一致してはなりません。

以下の例は、アドレス範囲書式を示しています。

0-9,100-1FF	2 つの範囲。1 つは 0 から 9F まで、1 つは 100 から 1FF まで
[1000-1FFF]*3+2000	3 つの範囲。 1000-1FFF，3000-3FFF，5000-5FFF
[1000-1FFF]*3	3 つの範囲。 1000-1FFF，2000-2FFF，3000-3FFF
[50-77F]/200	5 つの範囲。 50-1FF，200-3FF，400-5FF，600-77F

セグメント配置コマンドライン・オプションのすべての数値は、先頭に .（ピリオド）がつけられていない場合には、16 進と解釈されます。すなわち、10 および .16 と記述された数値は、ともに十進で 16 と解釈されます。

メモリ・マップにセグメントを割り付ける方法と割り付ける場所を指定するには、(-Z)を使用します。

リンカが(-P)、(-Z)あるいは(-b)（バンク定義コマンド）のどちらにも定義されていない入力ファイルのセグメントを検出した場合、リンカはエラーを表示します。

far メモリ（FAR、FARCODE、FARCONST セグメント型）への配置は、分離して行われます。(-Z)オプショ

ンを far メモリに用いると、最初のページと範囲に全体が収まるセグメントを連続的に配置し、次に残りをパックされた配置で配置します。このことは、セグメントが(-P)オプションのような方法で、新たなパックされたセグメントにより置き換えられるかも知れないことを意味します。

コマンドライン用 バンク・セグメントを定義する (-b)

構文： -b [addrtype] [(type)]
bankedsegments = first, length, increment[, count]

ここで、パラメータは以下のとおりです。

addrtype	コードのダンプ時に使用されるロード・アドレスの型
	省略 バンク番号付きの論理アドレス
	# 線形物理アドレス
	@ 64180 タイプ物理アドレス
type	ターゲット・プロセッサに適用可能な場合は、すべてのセグメントにメモリ・タイプを指定します。このパラメータを省略すると、デフォルトの UNTYPED が使用されます。
bankedsegments	リンクするバンクに配置されるセグメントのリスト
	リストの中のセグメント間のデリミタは、セグメントのパック方法を決定します。
	: (コロン) 次のセグメントは、新しいバンクに置かれます。
	, (カンマ) 次のセグメントは、(できれば)前のセグメントと同じバンクに置かれます。
first	バンク・セグメント・リストの中の最初のセグメントの開始アドレス。これは 32 ビット値です。上位 16 ビットは先頭バンク番号を、下位 16 ビットは論理アドレス領域のバンクの開始アドレスを表します。
length	バイト数で表された各バンクの長さ。これは、16 ビット値です。
increment	バンク間の増分係数です。すなわち、次のバンクを得るために first に追加される番号です。これは 32 ビット値です。上位 16 ビットはバンクのインクレメントを、下位 16 ビットは論理アドレス領域の開始アドレスからのインクレメントを表します。
count	利用可能なバンク数を 10 進で指定します。

バンク切替え操作用に設計されたプログラムにバンク・セグメントを割り付けるには、-b を使用します。またこのオプションを使用すると、リンカ操作をバンキング・モードにします。

2つ以上の-b 定義ができます。

論理アドレスは、プログラムで参照されるアドレスです。大抵のバンク切替えスキームにおいては、これは、論理アドレスが、最上位の 16 ビットにバンク番号を、最下位の 16 ビットにオフセットを含むことを意味します。

リニア物理アドレスは、バンク番号 (アドレスの最上位 16 ビット) にバンク長をかけ、オフセット (アドレスの最下位 16 ビット) を加えて計算されます。リニア物理アドレスの指定は、XLINK で出力されるバイトアドレスのロードに影響を与えますが、プログラムで参照されるアドレスには影響を与えません。

64180 タイプの物理アドレスは、バンク番号の最下位 8 ビットをとり、これを左に 12 ビットシフトし、さらにオフセットを加えて計算されます。

これらの単純な変換のいずれかを使用することは、他のいくらか単純なメモリレイアウトにたいしてのみ有益です。XLINK で計算されるようなリニア物理アドレスは、アドレス空間のまさに終わりにあるバンクメモリに対して有益です。より複雑なものについては、XLINK 出力への、PROM プログラマまたは特殊なプログラムによる、いくらかの後処理が必要とされます。そのようなプログラムのスタートについては、simple サブディレクトリを参照してください。

たとえば、アドレス 8000 から始まり、バンクの長さが 4000 で、バンク間インクレメントが 10000 となるように、3個のコード・セグメント BSEG1，BSEG2，BSEG3 をバンクにリンクするよう指定するには、次のようにします。

```
-b (CODE) BSEG1,BSEG2,BSEG3=008000,4000,10000
```

割付セグメントの種類

下表は、XLINK で処理することができる異なった種類のセグメントを一覧します。

セグメントの型	内容
STACK	デフォルトでは、上位アドレスから下位アドレスへ割り付けられます。整合されたセグメント・サイズは、割付前にロード・アドレスから差し引かれ、後続のセグメントは先行セグメントの下に置かれます。
RELATIVE COMMON	デフォルトでは、下位アドレスから上位アドレスへ割り付けられます。

スタック・セグメントがセグメント定義の中で相対セグメントやコモン・セグメントとミックスされた場合、リンカは警告メッセージを発生しますが、セグメント・リストの第1セグメントによって設定されたデフォルトの割付に従ってセグメントを割り付けます。

コモン・セグメントは、同名のセグメントの中に最大の宣言に等しいサイズをもっています。すなわち、モジュールAがサイズ4のコモン・セグメント COMSEG を宣言し、モジュールBがこのセグメント(COMSEG)をサイズ5で宣言した場合、セグメントには後者のサイズ(5バイト)が割り付けられます。

コードや初期化演算子をもつコモン・セグメントをオーバレイしないように注意してください。

相対セグメントやスタック・セグメントは、異なった (整合された) 宣言の和に等しいサイズをもちます。

セグメントのメモリ・タイプ

リスト内の全てのセグメントに型を割り当てるには、任意指定の type パラメータを使用します。type パラメータは、XLINK のセグメントの重なりの処理方法に効果があります。さらにこのパラメータは、一部のハードウェア・エミュレータや SW シミュレータで使用される一部の出力形式で情報を生成します。

セグメント型	内容
BIT	ビット・メモリ*
CODE	コード・メモリ
DATA	データ・メモリ
FAR	FAR メモリのデータ。XLINK は、FAR メモリへのアクセスのチェックは行いません。64KB の境界をまたぐセグメントの一部は上方向に移され、境界から始まるようになる。
FARC，FARCONST	(上記のように動作する) FAR メモリの定数
FARCODE	FAR メモリ内のコード
HUGE	HUGE メモリのデータ。境界をまたぐ問題はない。
HUGEC，HUGECONST	HUGE メモリの定数
HUGECODE	HUGE メモリ内のコード
IDATA	内部データメモリ
NEAR	NEAR メモリのデータ。16 ビット・アドレシングでアクセスされる。このセグメントは 32 ビットアドレス空間のどこにでも配置される
NEARC，NEARCONST	NEAR メモリの定数
NPAGE	絶対アドレス指定によってアクセスされるデータ・メモリ
UNTYPED	デフォルトの型
XDATA	外部データメモリ
ZPAGE	ゼロページ・データ・メモリ

*BIT セグメントのアドレスは、バイトではなくてビットで指定されます。BIT メモリは最初に割り当てられます。

範囲エラー

-Z コマンドで指定された範囲が短すぎる場合に、あるセグメントが別のセグメントにオーバーラップなっているのであればエラー24 Segment segment overlaps segment segment が、また範囲が小さすぎるのであればエラー26 Segment segment is too long が発生します。

デフォルトでは、XLINK はチェックを行い、(-Z コマンドと絶対セグメントによって) 定義された各種のセグメントがメモリで重ならないようにします。

例

アドレス0に SEGA を位置づけし、すぐに SEGB を続けるには、次のようにします。

```
-Z(CODE)SEGA,SEGB=0
```

SEGA を FFFH から下向き方向に割り付け、その下に SEGB を続けるには、次のようにします。

```
-Z(CODE)SEGA,SEGB#FFF
```

メモリの特定の領域を SEGA，SEGB に割り付けるには、次のようにします。

```
-Z(CODE)SEGA,SEGB=100-1FF,400-6FF,1000
```

この例では、SEGA がアドレス 100 と 1FF の容量にフィットするならば、SEGA はその間の領域に置かれます。この領域に適合しない場合、XLINK はアドレス 400～6FF の範囲を試みます。これらの範囲のいずれもが SEGA を保持するのに十分でない場合、SEGA はアドレス 1000 から始まります。

SEGB は、セグメント SEGA の後に上記と同じ規則に従って置かれます。SEGA がアドレス 100～1FF の範囲にフィットする場合、XLINK は（SEGA に続けて）SEGB もその領域に置こうと試みます。SEGB の容量が大きすぎる場合、SEGB はアドレス 400～6FF の範囲に置かれます。それでも適合しないようならば、SEGB はアドレス 1000 から始められます。

```
-Z(NEAR)SEGA,SEGB=19000-1FFFF
```

セグメント SEGA と SEGB はアドレス 19000～1FFFF に置かれますが、デフォルトの 16 ビット・アドレシング・モードがデータ（すなわち、9000～FFFF）をアクセスするのに使用されます。

第１１章 XLINK 出力形式

この章では、XLINK 出力形式の概要をまとめます。

単一出力ファイル

次の形式では、出力ファイルが1個生成されます。

形式	型	拡張子	アドレス型
AOMF8051†	バイナリ	CPU から	N
AOMF8096†	バイナリ	CPU から	N
AOMF80196†	バイナリ	CPU から	N
AOMF80251	バイナリ	CPU から	N
ASHLING	バイナリ	なし	N
ASHLING-6301†	バイナリ	CPU から	N
ASHLING-64180†	バイナリ	CPU から	NS
ASHLING-6801†	バイナリ	CPU から	N
ASHLING-8080†	バイナリ	CPU から	NS
ASHLING-8085†	バイナリ	CPU から	NS
ASHLING-Z80†	バイナリ	CPU から	NS
DEBUG (UBROF) †§	バイナリ	.dbg	NL
ELF	バイナリ	.elf	NL
EXTENDED-TEKHEX†	ASCII	CPU から	NLPS
HP-CODE	バイナリ	.x	NLPS
HP-SYMB	バイナリ	.l	NLPS
IEEE695 †**	バイナリ	.695	NL
INTEL-STANDARD	ASCII	CPU から	N
INTEL-EXTENDED	ASCII	CPU から	NLPS
MILLENIUM (Tektronix)	ASCII	CPU から	N
MOTOROLA	ASCII	CPU から	NLPS
MPDS-CODE	バイナリ	.tsk	N
MPDS-SYMB	バイナリ	.sym	NLPS

形式	型	拡張子	アドレス型
MSD	ASCII	.sym	N
MSP430_TXT	ASCII	.txt	NLPS
NEC-SYMBOLIC†	ASCII	.sym	N
NEC2-SYMBOLIC†	ASCII	.sym	N
NEC78K-SYMBOLIC†	ASCII	.sym	N
PENTICA-A	ASCII	.sym	NLPS
PENTICA-B	ASCII	.sym	NLPS
PENTICA-C	ASCII	.sym	NLPS
PENTICA-D	ASCII	.sym	NLPS
RCA	ASCII	CPU から	N
SIMPLE	バイナリ	.raw	NLPS
SYMBOLIC	ASCII	CPU から	NLPS
SYSROF†	バイナリ	.abs	NLPS
TEKTRONIX (Millenium)	ASCII	.hex	N
TI7000 (TMS7000)	ASCII	CPU から	N
TYPED	ASCII	CPU から	NLPS
UBROF†	バイナリ	.dbg	NL
UBROF5†	バイナリ	.dbg	NL
UBROF6†	バイナリ	.dbg	NL
UBROF7†	バイナリ	.dbg	NL
XCOFF78k	バイナリ	.lnk	NL
ZAX	ASCII	CPU から	NLPS

†付きの形式は、セグメントの型宣言によって異なります。すなわち、XLINK -Z オプションで指定される type フィールドが重要になります。

** CPU とデバッガの特定の組み合わせに対してのみサポートされます。詳細な情報については、XLINK.TXT および XMAN.TXT を参照してください。

§ -FUBROF（または –FDEBUG）を使用すると、入力の最新 UBROF 形式バージョンに一致する UBROF 出力を生成します。-FUBROF5（または –FUBROF6）は、入力に関係なく、指定されたバージョンの形式を強制的に出力します。

アドレス型

アドレス型は、次の1つになります。

N＝非バンク化アドレス

L＝バンク化論理アドレス

P＝バンク化物理アドレス

S＝バンク化 64180 物理アドレス

２つの出力ファイル

次の形式では、出力ファイルが2個生成されます。

形式	コード形式	拡張子	シンボル形式	拡張子
DEBUG-MOTOROLA	DEBUG	.axx	MOTOROLA	.obj
DEBUG-INTEL-EXT	DEBUG	.axx	INTEL-EXT	.hex
DEBUG-INTEL-STD	DEBUG	.axx	INTEL-STD	.hex
HP	HP-CODE	.x	HP-SYMB	.1
MPDS	MPDS-CODE	.tsk	MPDS-SYMB	.sym
MPDS-I	INTEL-STANDARD	.hex	MPDS-SYMB	.sym
MPDS-M	MOTOROLA	.s19	mpds-symb	.SYM
MSD-I	INTEL-STANDARD	.hex	MSD	.sym
MSD-M	MOTOROLA	.hex	MSD	.sym
MSD-T	MILLENIUM	.hex	MSD	.sym
NEC	INTEL-STANDARD	.hex	NEC-SYMB	.sym
NEC2	INTEL-STANDARD	.hex	NEC2-SYMB	.sym
NEC78K	INTEL-STANDARD	.hex	NEC2-SYMB	.sym
PENTICA-AI	INTEL-STANDARD	.obj	PENTICA-A	.sym
PENTICA-AM	MOTOROLA	.obj	PENTICA-A	.sym

形式	コード形式	拡張子	シンボル形式	拡張子
PENTICA-BI	INTEL-STANDARD	.obj	PENTICA-B	.sym
PENTICA-BM	MOTOROLA	.obj	PENTICA-B	.sym
PENTICA-CI	INTEL-STANDARD	.obj	PENTICA-C	.sym
PENTICA-CM	MOTOROLA	.obj	PENTICA-C	.sym
PENTICA-DI	INTEL-STANDARD	.obj	PENTIDA-D	.sym
PENTICA-DM	MOTOROLA	.obj	PENTIDA-D	.sym
ZAX-I	INTEL-STANDARD	.hex	ZAX	.sym
ZAX-M	MOTOROLA	.hex	ZAX	.sym

出力形式の異形

Format variant (-Y)オプションを用い、指定された出力形式に対して以下の拡張が選択できます。

出力形式	オプション	説明
PENTICA-A,B,C,D	Y0	modules:symbolname のようなシンボル。
および MPDS-SYMB	Y1	module:symbolname のようなラベルと行。
	Y2	module:symbolname のような行。
AOMF8051	Y0	Hitex 用の特殊な型の情報。
INTEL-STANDARD	Y0	00000001FF でのみ終了する。
	Y1	PGMENTRY で終了。それ以外は 0000001FF。
MPDS-CODE	Y0	代わりに 0XFF でみたす。
DEBUG,-r	Y#	古い UBROF のバージョン。
INTEL-EXTENDED	Y0	セグメント・バリアント
	Y1	32 ビット・リニア・バリアント
ELF	Y0	DWARF デバッグで出力
	Y1	DWARF デバッグ出力を抑制する。

本マニュアル作成以降の利用可能な追加オプションについては、XLINK.TXT ファイルを参照してください。

いくつかの形式に対して出力形式の異形（バリアント）を指定するには、Format variant (-y)を使用します。フラッグキャラクタは、新しいオプション-y のあとに指定できます。影響される形式は、IEEE695、ELF、および XCOFF78K です。

IEEE695

IEEE695 に対して利用可能な形式の異形は、以下のように指定されます。

指定子	説明
#define 定数なし（-yd）	いかなる#define 定数レコードも出力しません。これは出力ファイルのサイズをしばしば劇的に削減します。
グローバル型をグローバルに出力（-yg）	出力ファイルの最初にある BB2 ブロック内で可視的な型をグローバルに出力します。
グローバル型をそれぞれのモジュール内で出力（-yl）	出力ファイルのそれぞれのモジュールの最初にある BB1 ブロック内で可視的な型をグローバルに出力します。
ビット・セクションをバイト・セクションとして扱う（-yb）	XLINK は、ビット変数を表す IEEE695 に基づく変数をサポートし、ビットアドレシング可能なセクション用にビットアドレスを使用します。この指定子をオンにすると、XLINK はこれら変数とセクションを、あたかもバイトであるかのように扱います。
出力を三菱 PDB30 デバッガ用に調節する（-ym）	この指定子をオンにすると、出力を三菱 PDB30 デバッガ用にいくらか特別な方法で調節します。（注意）l および b 指定子は同様に使う必要があります。（-ylbm）
ブロック内でローカルな定数なし（-ye）	この指定子を使用すると、XLINK はいかなるブロック内ローカルな定数も出力ファイルに出力しません。これが起こりうる 1 つの状況としては、enum がブロック内で宣言された場合が挙げられます。
変数の寿命を扱う（-yy）	位置が変化する変数に対するより正確なデバッグ情報出力するには、IEEE695 における変数の寿命のサポートを使用します。
スタック調整レコードを出力する（-ys）	仮想フレーム・ポインタのスタック・ポインタからのオフセットを示すため、IEEE695 スタック調整レコードを出力します。

指定子	説明
BB10 ブロック内でローカルなモジュールを出力する (-ya)	もしあれば、BB3 (高レベル) ブロックと同様に、BB10 (アセンブラ・レベル) ブロック内のモジュール・ローカル・シンボルについての情報を出力します。
最後のリターンがエンド・オブ・ファンクションを参照する(-yr)	関数内の最後のリターン文の行に対するソース行情報を変更し、その関数の最終行を現在の行の代わりに参照するようにします。

デバッガごとに推奨される形式の異形の指定子は、以下のように与えられます。

デバッガ	指定子
6812 Noral デバッガ	-ygvs
68HC16 Microtek デバッガ	-ylb
740 三菱デバッガ	-ylmba
7700 HP RTC デバッガ	-ygb
7700 東芝 RTE900 m25	-ygbe
H8300 HP RTC デバッガ	-ygb
H8300H HP RTC デバッガ	-ygb
H8S HP RTC デバッガ	-ygb
M16C HP RTC デバッガ	-ygb
M16C 三菱 PDB30 デバッガ	-ylbms
T900 東芝 RTE900 m25	-ygbe

ELF

ELF に対して利用可能な形式の異形は、以下のように指定されます。

指定子	説明
DWARF デバッグ出力を抑制する (-yn)	デバッグ情報なしの ELF ファイルを出力します。(注意) ELF 形式は現在のところ、68HC12 および V850 ターゲット・プロセッサに対してサポートされています。

XCOFF78K

XCOFF78K に対して利用可能な形式の異形は、以下のように指定されます。

指定子	説明
-ys	シンボルを 31 文字に切りつめる。
-yp	ソースファイルのパスを取り除く。
-ye	モジュール enum を含む。
-yl	行番号情報を引きずる。

2 つ以上の指定を行うには、これらはすべて同じ-y オプションのあとに指定される必要があります。たとえば、s と p の指定子を両方用いるには、 -ysp にようにします。

第１２章 XLIB ライブラリアン

この章では、再配置可能ライブラリ・ルーチンを作成、維持することができるように作成された XLIB ライブラリアンを説明します。

概要

XLINK リンカのように、XLIB ライブラリアンは UBROF (Universal Binary Relocatable Object Format、汎用バイナリ再配置可能オブジェクト形式) 標準オブジェクト形式を使用し、広範な 32 ビット・バイト向きプロセッサ (現在の主要なマイクロプロセッサのほとんどに適用されています) のサポートが可能になっています。

ライブラリ

ライブラリとは多数の再配置可能オブジェクト・モジュールをもった単一ファイルであり、各モジュールは必要なときにファイルの他のモジュールから個別にロードすることができます。

通常、ライブラリ・ファイルのモジュールはすべて LIBRARY 属性をもっています。つまり、モジュールはプログラムで実際に必要な場合にのみ、リンカによってロードされるというわけです。これは、モジュールのデマンド・ローディングと言います。

一方、PROGRAM 属性をもつモジュールは、このモジュールをもつファイルがリンカで処理されると必ずロードされます。

ライブラリ・ファイルは、多数の LIBRARY 型モジュールを格納しているという点を除けば、アセンブラや C/EC++コンパイラによって生成される他の再配置可能オブジェクト・ファイルとなんら変わりありません。

Ｃプログラムをもつライブラリの使用法

全ての C プログラムはライブラリを活用します。C/EC++コンパイラには、多数の標準ライブラリ・ファイルが添付されています。

ほとんどの C/EC++プログラマは、次のような理由のいずれか 1 つにより、ある時点で XLIB ライブラリアンを使用します。

- 標準ライブラリの 1 つにあるモジュールを置換あるいは変更するため。例えば、ライブラリアンは、CSTARTUP および／または putchar モジュールの配布バージョンをカスタマイズ化したモジュールで置き換える場合に使用されます。

- C/EC++プログラムがリンクされるたびにC/EC++やアセンブラ・モジュールが常に利用できるように、標準ライブラリ・ファイルにC/EC++やアセンブラ・モジュールを追加するため。
- 必要に応じて標準Cライブラリと一緒にプログラムにリンクすることができるカスタム・ライブラリ・ファイルを生成するため。

アセンブラ・プログラムをもつライブラリの使用法

アセンブラだけを使用する場合は、ライブラリを使用する必要はありません。しかしながら、ライブラリは、特に中規模から大規模のアセンブラ・アプリケーションを記述する場合に次のような利点を与えてくれます。

- 2つ以上のプロジェクトで使用するユーティリティ・モジュールを単純なライブラリ・ファイルに組み合わせることができます。これによって、必要な全てのモジュールに入力ファイルのリストを含む必要性がなくなり、リンク処理が簡潔になります。プログラムに必要なライブラリ・モジュールのみが出力ファイルに組み込まれます。
- 複数のモジュールを単一のアセンブラ・ソース・ファイルに置くことができるため、プログラムの保守が容易になります。
- アプリケーション、保守、ドキュメント化を構成するオブジェクト・ファイルの数を削減することができます。

次の2つの基本方法のいずれかを使用して、アセンブリ言語ライブラリ・ファイルを生成することができます。

- ライブラリ・ファイルは、複数のライブラリ型モジュールを含む単一のアセンブラ・ソース・ファイルをアセンブルすることによって生成することができます。結果のライブラリ・ファイルは、XLIBを使用して変更できます。
- ライブラリ・ファイルは、XLIB ライブラリアンを使用して好きな数の既存のモジュールをマージさせ、ユーザ生成ライブラリを形成することによって生成できます。

NAME や MODULE アセンブリ疑似命令は、それぞれモジュールを PROGRAM 型や LIBRARY 型として宣言するために使用されます。

第１３章 XLIB コマンドの概要

この章では、機能によって分類されたライブラリアン･コマンドの概要をまとめます。
次の章に、ライブラリアン･コマンドをアルファベット順に並べた参照リストを示します。

ライブラリ・リスト・コマンド

LIST-ALL-SYMBOLS	モジュールのシンボルをリストします。
LIST-CRC	モジュールの CRC 値をリストします。
LIST-DATE-STAMPS	モジュールの日付をリストします。
LIST-ENTRIES	モジュールの PUBLIC シンボルをリストします。
LIST-EXTERNALS	モジュールの EXTERN シンボルをリストします。
LIST-MODULES	モジュールをリストします。
LIST-OBJECT-CODE	低レベルの再配置可能コードをリストします。
LIST-SEGMENTS	モジュールのセグメントをリストします。

ライブラリ編集コマンド

DELETE-MODULES	ライブラリからモジュールを取り外します。
FETCH-MODULES	ライブラリにモジュールを追加します。
INSERT-MODULES	ライブラリのモジュールを移動します。
MAKE-LIBRARY	モジュールをライブラリ型に変更します。
MAKE-PROGRAM	モジュールをプログラム型に変更します。
RENAME-ENTRY	PUBLIC シンボルの名前を変えます。
RENAME-EXTERNAL	EXTERN シンボルの名前を変えます。
RENAME-GLOBAL	EXTERN シンボル、PUBLIC シンボルの名前を変えます。
RENAME-MODULE	1つ以上のモジュールの名前を変えます。
RENAME-SEGMENT	1つ以上のセグメントの名前を変えます。
REPLACE-MODULES	実行可能コードをアップデートします。

その他のライブラリ・コマンド

CD	ワーキング・ディレクトリを変更します。
COMPACT-FILE	ライブラリ・ファイルのサイズを縮小します。
DEFINE-CPU	ＣＰＵタイプを指定します。
DIRECTORY	利用可能なオブジェクト・ファイルを表示します。
DISPLAY-OPTIONS	XLIB オプションを表示します。
ECHO-INPUT	コマンド・ファイル診断ツール
EXIT	オペレーティング・システムに戻ります。
EXTENSION	デフォルトの拡張子を定義します。
HELP	ヘルプ情報を表示します。
ON-ERROR-EXIT	バッチ・エラーが発生すると終了します。
PWD	ワーキング・ディレクトリを出力します。
QUIT	オペレーティング・システムに戻ります。
REMARK	コマンド・ファイルのコメント

第１４章 XLIB コマンドのリファレンス

この章では、すべてのライブラリアン･コマンドの構文と機能を詳細に説明します。

識別子の個々のワードは、不明瞭にならない範囲で省略することができます。例えば、
LIST-MODULES は L-M と省略できます。

XLIB を実行している際は、いつでもリターンを押して、情報を求めたり、オプション･リストを表示させたりできます。

コマンドラインからの XLIB コマンド

コマンドライン･スイッチ-c を使用すると、XLIB コマンドをコマンドラインから実行できます。-c スイッチのあとのそれぞれのコマンドライン引数は、1 つのコマンドとして用いられます。たとえば、

```
xlib -c "list-mod math.r31 m.txt" "list-mod mod.r31 m.txt"
```

は、XLIB 中で以下のように入力することと同等です。

```
*list-mod math.r31 m.txt
*list-mod mod.r31 m.txt
*QUIT
```

注意：それぞれのコマンドライン引数は、"で囲まれなければなりません。

XLIB バッチファイル

ファイルを指定した一つのコマンドライン･パラメータで XLIB を実行すると、XLIB はコンソールの代わりにそのファイルからコマンドを読みます。

パラメータ

次のパラメータは、多くの XLIB コマンドに共通のものです。

パラメータ	内容	
objectfile	オブジェクト・モジュールをもつファイル	
start, end	次の形式の1つで処理される最初と最後のモジュール	
	n	n番目のモジュール
	$	最後のモジュール
	name	モジュール名
	name+n	name から n モジュール後にあるモジュール
	$-n	最後のモジュールから n モジュールだけ前にあるモジュール
listfile	リスト・ファイル	
source	モジュールが読み取られるファイル	
destination	モジュールの出力先のファイル	

モジュールの式

ほとんどの XLIB コマンドでは、ソース・モジュール (RENAME-MODULE の oldname のように) やモジュールの範囲 (startmodule, endmodule) を指定することができます。また、指定しなければならないことがあります。

内部的に、全ての XLIB 操作では、モジュールは1から上向き方向に番号づけされます。モジュールは、モジュールの実際の名前で、モジュール名プラスまたはマイナスの相対式で、あるいは絶対番号で参照されることがあります。モジュール名が非常に長かったり、不明であったり、あるいは (空白やカンマなどの) 特異な文字を含んでいる場合は、後者が非常に有用になります。以下に、モジュール式で利用できる可変部のリストを示します。

モジュール名	内容内容
3	3番目のモジュール
$	最後のモジュール
name+4	*name* から4モジュール後にあるモジュール
name-12	*name* から 12 モジュール前にあるモジュール
$-2	最後のモジュールから2モジュール前にあるモジュール

したがって、LIST-MOD FILE,,$-2 コマンドは、FILE の最後の3つのモジュールを端末にリストします。

リスト形式

LIST コマンドは、各シンボルが次の接頭語の1つをもつシンボルをリストします。

接頭語	内容
nn.Pgm	相対番号 *nn* をもつプログラム・モジュール
nn.Lib	相対番号 *nn* をもつライブラリ・モジュール
Ext	カレント・モジュールの外部シンボル
Ent	カレント・モジュールのエントリ
Loc	カレント・モジュールのローカル・シンボル
Rel	カレント・モジュールの標準セグメント
Stk	カレント・モジュールのスタック・セグメント
Com	カレント・モジュールのコモン・セグメント

CD

ワーキング・ディレクトリを変更します。

構文

CD directory

説明

ワーキング・ディレクトリを変更するには、CD を使用します。

例

以下のコマンドは、ワーキング・ディレクトリをサブディレクトリ lib に変更します。

```
CD lib
```

COMPACT-FILE

ライブラリ・ファイルのサイズを小さくします。

構文

COMPACT-FILE objectfile

説明

短い絶対レコードを、可変長さをもつより長いレコードに連結するには、COMPACT-FILE を使用します。これによって、ライブラリ・ファイルのサイズが約5%小さくなり、ローダ／リンカ処理中により少ない時間しか要さないライブラリ・ファイルを得ることができます。

例

以下のコマンドは、ファイル maxmin.r31 をコンパクトにします。

```
COMPACT-FILE maxmin
```

この結果、次のように表示されます。

```
20 byte(s) deleted
```

DEFINE-CPU

CPUタイプを指定します。

構文

```
DEFINE-CPU cpu
```

パラメータ

cpu　　ターゲット・プロセッサ

説明

このコマンドは、オブジェクト・ファイルの操作を行う前に発行してください。

例

以下のコマンドは、CPU を 740 として定義します。

```
DEF-CPU 740
```

DELETE-MODULES

ライブラリからモジュールを取り外します。

構文

```
DELETE-MODULES objectfile start end
```

説明

指定モジュールを削除するには、DELETE-MODULES を使用します。

例

以下のコマンドは、モジュール 2 をファイル math.r31 より削除します。

```
DEL-MOD math 2 2
```

DIRECTORY

利用可能なオブジェクト・ファイルを表示します。

構文

DIRECTORY [指定子]

説明

ターゲット・プロセッサに適用される型のファイルをすべて端末に表示するには、DIRECTORY を使用します。規則子を指定しない場合は、カレント・ディレクトリのファイルがリストされます。

例

以下のコマンドは、カレント・ディレクトリのオブジェクト・ファイルをリスト表示します。

DIR

次のように表示されます。

```
general     770
math        502
maxmin      375
```

DISPLAY-OPTIONS

XLIB オプションを表示します。

構文

DISPLAY-OPTIONS [listfile]

説明

listfile にこのバージョンの XLIB で認識される全てのCPU名をリストするには、DISPLAY-OPTIONS を使用します。CPU別のオブジェクト・ファイルのデフォルトのファイル型もリストされます。その後、全 UBROF タグのリストが出力されます。

例

オプションをファイル opts.lst にリストするには、

DISPLAY-OPTIONS opts

ECHO-INPUT

コマンド・ファイル診断ツール

構文

ECHO-INPUT

説明

コマンド・ファイルは全てのコマンド入力を端末に表示させますので、ECHO-INPUT はコマンド・ファイルをバッチ・モードでデバッグする場合に有用になります。対話型モードでは、なんら効果はありません。

例

バッチファイル

ECHO-INPUT

は、あとに続くすべての XLIB コマンドをエコーします。

EXIT

オペレーティング・システムに戻します。

構文

EXIT

説明

対話形式のセッションの後に XLIB を終了するには、EXIT を使用します。

例

XLIB から抜けるには、

EXIT

EXTENSION

デフォルトの拡張子を設定します。

構文

EXTENSION

説明

デフォルトの拡張子を設定するには、EXTENSION を使用します。

FETCH-MODULES

ライブラリにモジュールを追加します。

構文

FETCH-MODULES source destination [start] [end]

説明

指定モジュールを destination ファイルに付加するには、FETCH-MODULES を使用します。既に destination ファイルが存在する場合、このファイルは空か、あるいは有効なオブジェクト・モジュールをもっていなければなりません。そうでない場合は、生成されます。

例

以下のコマンドは、モジュール mean を math.r31 から general.r31 にコピーします。

FETCH-MODULES math general mean

HELP

ヘルプ情報を表示させます。

構文

HELP [command] [listfile]

パラメータ

command　ヘルプ情報が表示されるコマンド

説明

パラメータを設定しないで HELP コマンドを指定した場合は、利用可能なコマンドのリストが端末に表示されます。パラメータを指定した場合は、パラメータと一致するすべてのコマンドが、その構文や機能の簡単な説明とともに表示されます。*は、すべてのコマンドと整合します。HELP 出力は、どのファイルにでも出力できます。

例

たとえば、コマンド

```
HELP LIST-MOD
```

は、以下の表示を行います。

```
LIST-MODULES <Object file> [<List file>] [<Start module>]
[<End module>]
    List the module names form [<Start module>] to
    [<End module>]
```

INSERT-MODULES

ライブラリのモジュールを移動します。

構文

```
INSERT-MODULES objectfile start end {BEFORE | AFTER} dest
```

説明

dest 前後の指定モジュールを移動するには、INSERT-MODULES を使用します。

例

以下のコマンドは、ファイル math.r31 内のモジュール mean をモジュール min の前に移動します。

```
INSERT-MOD math mean mean BEFOR min
```

LIST-ALL-SYMBOLS

モジュールの全シンボルをリストします。

構文

```
LIST-ALL-SYMBOLS objectfile [listfile] [start] [end]
```

説明

objectfile の中の指定モジュールの全シンボル（モジュール名、セグメント、外部、エントリ、ローカル）をリストするには、LIST-ALL-SYMBOLS を使用します。シンボルは、listfile にリストされます。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.rxx 内すべてのシンボルをリストします。

```
LIST-ALL-SYMBOLS math.rxx
```

これは、以下のような表示を行います。

```
          1. Lib max
               Rel   CODE
               Ent   max
               Loc   A
               Loc   B
               Loc   C
               Loc   ncarry
          2. Lib mean
               Rel   DATA
               Loc   CODE
               Loc   max
               Loc   A
               Loc   B
               Loc   C
               Loc   main
               Loc   start
          3. Lib min
               Rel   CODE
               Ent   min
               Loc   carry
```

LIST-CRC

モジュールの CRC 値をリストします。

構文

```
LIST-CRC objectfile [listfile] [start] [end]
```

説明

指定モジュールのモジュール名と関連 CRC 値をリストするには、LIST-CRC を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のすべてのモジュールに対する CRC をリストします。

```
LIST-CRC math
```

これは、以下の表示を行います。

```
        EC41                1.  Lib max
        ED72                2.  Lib mean
        9A73                3.  Lib min
```

LIST-DATE-STAMPS

モジュールの日付をリストします。

構文

```
LIST-DATE-STAMPS objectfile [listfile] [start] [end]
```

説明

指定モジュールのモジュール名とモジュール生成日をリストするには、LIST-DATE-STAMPS を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のすべてのモジュールに対する日付スタンプをリストします。

```
LIST-DATE-STAMPS math
```

これは以下の表示を行います。

```
09/Jan/99          1.  Lib  max
09/Jan/99          2.  Lib  mean
09/Jan/99          3.  Lib  min
```

LIST-ENTRIES

モジュールの PUBLIC シンボルをリストします。

構文

LIST-ENTRIES objectfile [listfile] [start] [end]

説明

指定モジュールのモジュール名と関連エントリをリストするには、LIST-ENTRIES を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のすべてのモジュールのエントリをリストします。

LIST-ENTRIES math

これは以下の表示を行います。

```
1.  Lib max
      Ent  max
2.  Lib  mean
3.  Lib  min
      Ent  min
```

LIST-EXTERNALS

モジュールの EXTERN シンボルをリストします。

構文

LIST-EXTERNALS objectfile [listfile] [start] [end]

説明

指定モジュールのモジュール名と関連外部シンボルをリストするには、LIST-EXTERNALS を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のすべてのモジュールの外部シンボルをリストします。

```
LIST-EXT math
```

これは以下の表示を行います。

```
          1.  Lib  max
          2.  Lib  mean
                Ext  max
          3.  Lib  min
```

LIST-MODULES

モジュールをリストします。

構文

```
LIST-MODULES objectfile [listfile] [start] [end]
```

説明

指定モジュールのモジュール名をリストするには、LIST-MODULES を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のすべてのモジュールをリストします。

```
LIST-MOD math
```

これは以下の表示を行います。

```
          1.  Lib  max
          2.  Lib  mean
          3.  Lib  min
```

LIST-OBJECT-CODE

低レベルの再配置可能コードをリストします。

構文

LIST-OBJECT-CODE objectfile [listfile]

説明

objectfile のコンテンツを ASCII フォーマットで listfile にリストするには、LIST-OBJECT-CODE を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、math.r31 内のオブジェクト・コードを object.lst にリストします。

LIST-OBJECT-CODE math object

LIST-SEGMENTS

モジュールのセグメントをリストします。

構文

LIST-SEGMENTS objectfile [listfile] [start] [end]

説明

指定モジュールのモジュール名と関連セグメントをリストするには、LIST-SEGMENTS を使用します。

各シンボルは、接頭語で識別されます。185 ページの「リスト形式」を参照してください。

例

以下のコマンドは、ファイル math.r31 内のモジュール mean のセグメントをリストします。

LIST-SEG math,,mean mean

listfile パラメータをスキップするために 2 つのコンマを使用していることに注意してください。

これにより、以下の表示が行われます。

```
          2.  Lib   mean
                Rel    DATA
                Rel    CODE
```

MAKE-LIBRARY

モジュールをライブラリ型に変更します。

構文

MAKE-LIBRARY objectfile [start] [end]

説明

モジュール･ヘッダー属性を変更して、指定モジュールに条件付きでロードするには、MAKE-LIBRARY を使用します。

例

以下のコマンドは、main.r31 のすべてのモジュールをライブラリ･モジュールに変換します。

MAKE-LIB main

MAKE-PROGRAM

モジュールをプログラム型に変更します。

構文

MAKE-PROGRAM objectfile [start] [end]

説明

モジュール･ヘッダー属性を変更して、指定モジュールに無条件でロードするには、MAKE-PROGRAM を使用します。

例

以下のコマンドは、main.r31 のモジュール start をプログラム･モジュールに変換します。

MAKE-PROG main start

ON-ERROR-EXIT

バッチ・エラーが発生すると、終了します。

構文

ON-ERROR-EXIT

説明

エラーが検出された場合にライブラリアンを中止するには、ON-ERROR-EXIT を使用します。

このコマンドは、バッチ・モードでの使用に最適です。

例

以下のバッチファイルは、もし FETCH-MODULES コマンドが失敗に終わった場合には、アボートされます。

```
ON-EEROR-EXIT
FETCH-MODULES math new
```

PWD

ワーキング・ディレクトリを出力します。

構文

PWD

説明

ワーキング・ディレクトリを出力するには、PWD を使用します。

QUIT

オペレーティング・システムに戻します。

構文

QUIT

説明

操作を終了し、オペレーティング・システムに戻るには、QUIT を使用します。

例

XLIB より抜けます。

```
QUIT
```

REMARK

コマンド・ファイルのコメント

構文

REMARK テキスト

説明

コメントを含むには、REMARK を使用します。

例

以下の例は、XLIB コマンド・ファイルのコメントの使用法を示しています。

```
REM Now compact file
COMPACT-FILE math
```

RENAME-ENTRY

PUBLIC シンボルの名前を変更します。

構文

RENAME-ENTRY objectfile old new [start] [end]

説明

指定モジュールのすべてのエントリ名を old から new に変更するには、RENAME-ENTRY を使用します。

例

以下のコマンドは、math.r31 内のモジュール 2 から 4 に対するエントリを mean から average にリネームします。

```
RENAME-ENTRY math mean average 2 4
```

RENAME-EXTERNAL

EXTERN シンボルの名前を変更します。

構文

RENAME-EXTERNAL objectfile old new [start] [end]

説明

指定モジュールのすべての外部シンボルの名前を old から new に変更するには、RENAME-EXTERNAL を使用します。

例

以下のコマンドは、math.r31 内のすべての外部シンボル error から err にリネームします。

```
RENAME-EXT math error err
```

RENAME-GLOBAL

EXTERN、 PUBLIC シンボルの名前を変更します。

構文

RENAME-GLOBAL objectfile old new [start] [end]

説明

指定モジュールの全ての外部シンボル、あるいはエントリの名前を old から new に変更するには、RENAME-GLOBAL を使用します。

例

以下のコマンドは、math.r31 内のすべての外部シンボル、あるいはエントリを mean から average にリネームします。

```
RENAME-GLOBAL math mean average
```

RENAME-MODULE

1つ以上のモジュールの名前を変更します。

構文

RENAME-MODULE objectfile old new

説明

モジュールの名前を変更するには、RENAME-MODULE を使用します。old の名前をもつモジュールが2つ以上ある場合は、最初に検出されたモジュールの名前だけが変更されます。

例

以下のコマンドは、math.r31 内のモジュール mean を average にリネームします。

```
RENAME-MOD math mean average
```

RENAME-SEGMENT

1つ以上のセグメントの名前を変更します。

構文

```
RENAME-SEGMENT objectfile old new [start] [end]
```

説明

指定モジュールのすべてのセグメントの名前を old から new へ変更するには、RENAME-SEGMENT を使用します。

例

以下の例は、ファイル math.r31 内のすべての CODE セグメントを ROM にリネームします。

```
RENAME-SEG math CODE ROM
```

REPLACE-MODULE

実行可能なコードをアップデートします。

構文

```
REPLACE-MODULE source destination
```

説明

同じ名前をもつモジュールを source から destination へ置き換えるには、REPLACE-MODULE を使用します。全ての置換は、端末に記録されます。このコマンドの主なアプリケーションは、大容量のランタイム・ライブラリのアップデートなどです。

例

以下の例は、math.r31 内のモジュール newmath.r31 からのモジュールで置き換えます。

```
REPLACE-MOD math newmath
```

これは以下の表示を行います。

```
          Replacing module 'max'
          Replacing module 'mean'
          Replacing module 'min'
```

第１５章 アセンブラ診断

この章では、740 アセンブラのエラー・メッセージ、警告メッセージをリストします。XLINK リンカや XLIB ライブラリアンのエラー・メッセージの詳細については、「XLINK 診断」、「XLIB 診断」を参照してください。

概要

エラー・メッセージは、任意指定のリスト・ファイルとともに端末に出力されます。

全てのエラーは、完全な、そのままで明白なメッセージとして発行されます。例えば、次のとおりです。

```
          ADS     B,C
------------------^
"testfile.s31",4  Error[40]: bad instruction
```

エラー・メッセージは、誤りのあるソース行、障害箇所のポインタ、その後に続く診断／ソース行番号から成ります。インクルード・ファイルが使用される場合は、エラー・メッセージの前にソース行番号とカレント・ファイルの名前がつきます。

```
          ADS     B,C
------------------^
"subfile.h",4  Error[40]: bad instruction
```

アセンブラが生成するエラー・メッセージは、次の6項目に分類されます。

- コマンド行エラー
- アセンブリ警告メッセージ
- アセンブリ・エラー・メッセージ
- 致命的なアセンブリ・エラー・メッセージ
- メモリ・オーバーフロー・メッセージ
- アセンブラ内部エラー

コマンド行エラー

コマンド行エラーは、アセンブラが不良パラメータで呼び出されると発生します。最も一般的な状況は、ファイルがオープンできない、重複、スペルミス、コマンド行切替えの抜けなどです。エラー・メッセージは自明なものです。

アセンブリ警告メッセージ

アセンブリ警告メッセージは、アセンブラがプログラミング・エラーや抜けなどによって発生したと考えられる構成体を検出すると発生します。これらの警告メッセージは、「警告メッセージ」の項にリストします。

アセンブリ・エラー・メッセージ

アセンブリ・エラー・メッセージは、アセンブラが言語規則に違反した構成体を検出すると発生します。これらのエラー・メッセージは、「エラー・メッセージ」の項にリストします。

致命的なアセンブリ・エラー

致命的なアセンブリ・エラー・メッセージは、アセンブラが以降の処理は無意味であると判断するほどのユーザ・エラーを検出すると発生します。診断メッセージが発行されると、アセンブラは直ちに終了します。致命的なエラー・メッセージは、エラー・メッセージ・リストの中で「Fatal」と記していますので、区別できます。

メモリ・オーバーフロー・メッセージ

アセンブラはメモリ・ベースのプログラムですので、主メモリ容量が小さなシステムの場合やソース・ファイルの容量が非常に大きな場合には、メモリ容量が足りなくなる恐れがあります。これは、次の特殊なメッセージによって確認できるようになっています。

```
* * * ASSEMBLER OUT OF MEMORY * * *

Dynamic memory used: nnnnnn bytes
```

このような状況が発生した場合の処置は、システム・メモリを追加するか、あるいはソース・ファイルをより小さなモジュールに分割するかのどちらかです。しかしながら、1MB の RAM があれば、アセンブラの能力は適度なサイズのソース・ファイルに対して十分なはずです。

アセンブラ内部エラー

アセンブリ中には、何回も内部一貫性チェックが実行されます。これらのチェックが1つでも不合格になると、アセンブラは不良内容の簡単な説明を発行した後に終了します。通常、このようなエラーは発生しませんが、万一発生した場合は、技術サポート・グループに連絡してください。その場合、問題に関する全ての情報をお知らせください。できれば、内部エラーを発生したプログラムのコピー・ディスクも提供してください。

エラー・メッセージ

一般

以下のセクションは、一般的なエラー・メッセージをリストしています。

No.	エラー・メッセージ	処置
0	Invalid syntax	アセンブラが式をデコードできません。
1	Too deep #include nesting (max. is 10)	致命的。#include ファイルを入れ子にするアセンブラの限界を超えています。再帰的#includeが原因と考えられます。
2	Failed to open #oinclude file 'name'	致命的。#include ファイルをオープンすることができません。指定されたディレクトリにファイルが存在しません。-I 接頭語をチェックしてください。
3	Invalid #include file name	致命的。#include ファイル名は、<file>あるいは"file"と記述してください。
4	Unexpected end of file encountered	致命的。ファイルの終りが条件付きアセンブリ内で、また繰返し疑似命令の中で、あるいはマクロ展開中に検出されました。条件付きアセンブリの終りがないなどが原因と考えられます。
5	Too long source line (max. is 2048 characters) truncated	ソース行の長さがアセンブラの限界を超えています。
6	Bad constant	正当な数字ではない文字が検出されました。

No.	エラー･メッセージ	処置
7	Hexadecimal constant without digits	16 進定数の接頭語 0x あるいは 0X が 16 進数に続かない状態で検出されました。
8	Invalid floating point constat	浮動小数点定数が大きすぎるか、あるいは構文が無効になっています。
9	Too many errors encountered (>100).	
10	Space or tab expected	
11	Too deep block nesting (max is 50)	プリプロセッサ疑似命令の入れ子が深すぎます。
12	String too long (max is 509)	アセンブラ文字列の長さの限界を超えています。
13	Missing delimiter in literal or character constant	文字あるいはリテラル定数に終了デリミタ ’ や ” が使用されていません。
14	Missing #endif	#if, #ifdef あるいは#ifndef が検出されましたが、対応する#endif がありません。
15	Invalid character encountered: <char>; ignored	
16	Identifier expected	ラベル名やシンボル名を待っていました。
17	‘)’ expected	
18	No such pre-processor command: <command>	＃の後に未知の識別子が続いています。
19	Unexpected token fond in pre-processor line	引数部が読み取られた後も、プリプロセッサ行が空になっていません。
20	Argument to #define too long (max is 2048)	

No.	エラー･メッセージ	処置
21	Too many formal parameters for #define (max is 127)	
22	Macro paprameter <parameer> redefined	#define シンボルの仮パラメータが繰り返されています。
23	‘,’ or ‘)’ expected	
24	Unmatched #else, #endif or #elif	致命的。#if, #ifdef あるいは#ifndef が抜けています。
25	#error <error>.	#error 疑似命令を介した印字出力
26	‘(‘ expected	
27	Too many active macro parameters (max is 256)	致命的。プリプロセッサの限界を超えています。
28	Too many nexted parameterized macros (max is 50)	致命的。プリプロセッサの限界を超えています。
29	Too deep macro nesting (max is 100)	致命的。プリプロセッサの限界を超えています。
30	Actual macro parameter too long (max is 512)	単一マクロ (#define 内の) 引数は、ソース行の長さを超えることはできません。
31	Macro <macro> called with too many parameters	使用されたパラメータの数が、マクロ定義内の数を超えています。
32	Macro <macro< called with too few parameters	使用されたパラメータの数が、マクロ定義 (#define) 内の数よりも少ないです。
33	too many MACRO arguments	アセンブラ･マクロの数が 32 を超えました。
34	may not be redefined	アセンブラ･マクロは、再定義することができません。

No.	エラー・メッセージ	処置
35	no name on macro	ラベルを使用していないアセンブラ・マクロ定義が検出されました。
36	Illegal formal parameter in mcaro	識別子ではないパラメータが見つけられました。
37	ENDM or EXITM not in macro	マクロに入っていないにもかかわらず、ENDM あるいは EXITM 疑似命令が検出されました。
38	'>' expected but found end-of-line	＜が検出されましたが、対応する＞がありません。
39	END before start of module	モジュールの終了疑似命令に、対応 MODULE 疑似命令がありません。
40	bad instruction	ニモニック／疑似命令が存在しません。
41	bad label	ラベルは、A～Z，a～z，＿ あるいは？から始めてください。後続の文字は、A～Z、 a～z、０～９、＿ あるいは？を使用してください。ラベルには、定義済みシンボルと同じ名前をつけることはできません。
42	duplicate label	ラベルは既にラベル・フィールドに表示されたか、あるいは EXTERN として宣言されました。
43	illegal effective address	このニモニックでは、アドレス指定モード (オペランド) は使用できません。
44	',' expected	カンマがあるはずですが、検出されません。
45	name duplicated	RSEG，STACK あるいは COMMON セグメントの名前は、既に他のものに使用されています。
46	segment type expected	RSEG，STACK あるいは COMMON 疑似命令で：が検出されましたが、その後に続くセグメント型が有効なものではありません。

No.	エラー・メッセージ	処置
47	segment name expected	RSEG，STACK あるいは COMMON 疑似命令には、名前が必要です。
48	value out of range ‘<range>‘	値がその限界を超えました。
49	alignment already set	RSEG，STACK あるいは COMMON 疑似命令では、2 回以上の整合の設定はできません。代わりに ALIGN，EVEN あるいは ODD を使用してください。
50	undefined symbol; <symbol>	ラベル・フィールドにも、EXTERN あるいは sfr 宣言にもシンボルがありませんでした。
51	Can’t be both PUBLIC and EXTERN	シンボルは、PUBLIC あるいは EXTERN のいずれかで宣言されます。
52	EXTERN not allowed	このコンテキストでは、EXTERN シンボルを参照することはできません。
53	expression must be absolute	式には、再配置可能シンボルや外部シンボルを含むことはできません。
54	expression can not be forwad	アセンブラは、初めて式が検出されたときに式を解くことができなければなりません。
55	illegal size	式の最大サイズは、32 ビットです。
56	too many digits	値があて先のサイズを超えています。
57	unbalanced conditional assembly directives	条件付きアセンブリ IF あるいは ENDIF が抜けています。
58	ELSE without IF	条件付きアセンブリ IF が抜けています。
59	ENDIF without IF	条件付きアセンブリ IF が抜けています。
60	unbalanced structured assembly directives	構造化アセンブリ IF あるいは ENDIF が抜けています。

No.	エラー･メッセージ	処置
61	'+' or '-' expected	プラスまたはマイナス符号が抜けています。
62	Illegal operation on extern or public symbol	パブリック･シンボルまたは外部シンボルで不正な演算が使用されました。例えば、SET です。
63	Illegal operation on non-constant label	非定数シンボルを PUBLIC や EXTERN にすることはできません。
64	Extern or unsolved expression	式はアセンブリ時に解いてください。すなわち、外部参照を含むことはできません。
65	'=' expected	等価符号が抜けています。
66	Segment too long (max is <max>)	ASEG, RSEG, STACK あるいは COMMON セグメントの長さが、アドレス可能な長さを超えています。
67	Public did not appear in label field	シンボルが PUBLIC 宣言されましたが、同じ名前のラベルがソース･ファイルに見つかりません。
68	End of block-repeat without start	ENDR 疑似命令はありますが、繰返し疑似命令 REPT が検出されません。
69	Seegment must be relocatable	ASEG では、この操作はできません。
70	Limit exceeded: <error text>, value is: <value> (decimal)	値が、LIMIT 疑似命令で設定された限界を超えました。このエラーテキストは、ユーザにより、LIMIT 疑似命令に設定されます。
71	Symbol '<symbol>' has already been declared EXTERN	EXTERN を再度 EXTERN として宣言しようとしました。
72	Symbol '<smbol>' has already been declared PUBLIC	PUBLIC を再度 PUBLIC として宣言しようとしました。

No.	エラー・メッセージ	処置
73	End-of-modle missing	ENDMOD が見つけられる前に、PROGRAM あるいは MODULE 疑似命令が検出されました。
74	Expression must yield non-negative result	正の値が要求されているのに、式は負の数値を評価しました。
75	Repeat directive unbalanced	このエラーは、対応する ENDR をもたない REPT 疑似命令、あるいは対応する REPT をもたない ENDR 疑似命令が原因で発生しました。
76	End of repeat directive is missing	閉じている ENDR のない REPT 疑似命令が検出されました。
77	LOCALs not allowed in this context, (<symbol>)	ローカル・シンボルは、マクロ定義内で宣言してください。
78	End of macro expected	アセンブラ・マクロが宣言されていますが、マクロの終了がありません。
79	End of repeat expected	繰返し疑似命令の1つが実行可能な状態になっていますが、繰返しの終りが見つけられません。
80	End of conditional assembly expected	条件付きアセンブリが実行可能な状態になっていますが、if の終りがありません。
81	End of structured assembly expected	構造化アセンブリの疑似命令の1つが実行可能な状態になっていますが、対応する END がありません。
82	Misplaced end of structured assembly	構造化アセンブリ疑似命令の1つを終了する疑似命令が検出されましたが、対応する START 疑似命令が実行可能な状態になっていません。
83	Error in SFR attribute definition	SFRTYPE 疑似命令が未知の属性で使用されました。
84	Illegal symbol type in symbol <symbol>	このシンボル型は間違っているので、このコンテキストでは使用できません。
85	Wrong nr of arguments	

No.	エラー･メッセージ	処置
86	Number expected	数字以外のものに遭遇しました。
87	Label must be public or extern	ラベルが PUBLIC または EXTERN で宣言される必要があります。
88	Label 'label' not defined with DFFN	ラベルをこのコンテキストで使用する前に DFFN を通して定義する必要があります。
89	Sorry DEMO versions, bytecounte exceeded	
90	Different parts of ASEG has to overlapping code	
91	Internal error	
92	Empty macro stack	空のスタックでポップできません。
93	Macro stack overflow	100 以上の項目がプッシュされました。
94	Attempt to access out-of-stack value	
95	Invalid macro operator 'operator'	
96	No such macro argument	マクロをコールするパラメータ数が不足しています。
97	Sorry Lite version, bytecount exceeded (max bytes)	
98	Option –re cannot handle code in include files, use –r or –rn instead	
99	#include within macro not supported	

740 に特定されるエラー・メッセージ

一般エラーに加え、740 アセンブラは次のエラーを生成します。

400 Absolute operand is not possible here

401 Branch displacement out of range Valid range is -128 to 127.

402 Nothing to BREAK out of

403 Unterminated C-comment

404 CASE after DEFAULT

405 CASE outside SWITCH

406 COMMA expected

407 Nothing to CONTINUE to

408 Cannot solve break count value

409 DEFAULT outside SWITCH

410 ELSE used more than once

411 ELSE without matching IF

412 ELSEIF cannot be used after ELSE

413 ELSEIF with no matching IF

414 ENDF without matching FOR

415 ENDIF without matching IF

416 ENDS without matching SWITCH

417 ENDW without matching WHILE

418 Wrong syntax, file not included

419 Illegal option for OPT directive

420 Illegal suffix

421 Multiple DEFAULT

422 Expression must be absolute

423 Step value must be immediate

424 Step value in register not allowed here

425 Expression must be positive

426 DOWNTO not allowed when step value in register

427 Branch too long and optimisation turned off

428 Break argument must be 1,2, or 3

429 UNTIL without matching REPEAT

430 Accessing SFR incorrectly, check read/write flags

431 Accessing SFR using incorrect size

432 Address out of range. Valid range is 0 to 255.

433 Address out of range. Valid range is 0 to 65535.

434 Number out of range. Valid range is -128 to 255.

435 Bit-number out of range. Valid range is 0 to 7.

436 Can't branch to a negative address.

437 Step value must be #1, #2, #3 or #4, when using X or Y register

438 Register offset out of range. Valid range is -128 to 255.

439 Register offset out of range. Valid range is -32768 to 65536.

440 Address can't be negative.

441 Cannot solve value.

警告メッセージ

一般

以下のセクションは、一般的な警告メッセージをリストしています。

No.	警告メッセージ	処置
0	unreferenced label	ラベルがオペランドとして使用されていないか、あるいはパブリック宣言されています。
1	Nested comment	Cコメントが入れ子になっています。
2	unknown escape sequence	バックスラッシュ（\）が文字定数の中で検出されたか、あるいは文字列リテラルの後に未知のエスケープ文字が続いています。
3	Non-printable character	リテラルあるいは文字定数に印字不可能文字が検出されました。
4	Macro or define expected	
5	Floating point value out-of-range	浮動小数点値が、ターゲット・プロセッサの浮動小数点システムで表現するには大きすぎます。
6	Floating point division by zero	
7	Wrong usage of string operator ('#' or '##'); ignored	現行の処理系では、#および##演算子の用途は、パラメータ化されたマクロのトークン・フィールドに限定されています。さらに、#演算子はフォーマット・パラメータの前に指定される必要があります。
8	Macro parameter(s) not used	
9	Macro redefined	
10	Unknown macro	

No.	警告メッセージ	処置
11	Empty macro argument	
12	Recursive macro	
13	Redefinition of Special Function Register	SFR は既に定義済みです。
14	Division by zero	定数式に、0による除算があります。
15	Constant truncated	定数があて先のサイズよりも長すぎます。
16	Suspicious sfr expression	式に特殊関数レジスタ SFR が使用され、アセンブラがアクセス権をチェックできません。
17	Empty module ‘<module name>‘, module skipped	ENDMOD あるいは MODULE の直後に END を、そのあとに間に文をいれないで ENDMOD を続けて使用することによって、空モジュールが生成されています。
18	End of program while in include file	ファイルがインクルードされているにもかかわらず、プログラムが終了しました。
20	Bit symbol cannot be used as operand	シンボルが bit 疑似命令を使用して宣言されていますが、ビット･アドレスは計算されませんので、シンボルは使用すべきではありません。
21	Label did not appear in the label field	
22	Set segment alignment the same <value> or larger	ALIGN で設定された整合がセグメント整合より大きい場合、これはリンク時に失われます。

740 に特有の警告メッセージ

一般的な警告に加えて、740 アセンブラは以下の警告メッセージを発生します。

400 Byte mode not available for this mnemonic

401 Number out of range

402 Same register appears more than once

403 SFR neither defined as READ nor WRITE

404 More than one SFR size attribute defined, using default (byte)

405 No SFR size attribute defined, using default (byte)

第１６章 XLINK 診断

この章では、XLINK リンカが発生するエラー・メッセージと警告メッセージを説明します。

概要

XLINK リンカが発生するエラー・メッセージは、次の5つの項目に分類されます。

- リンカ警告メッセージ
- リンカ・エラー・メッセージ
- 致命的なリンカ・エラー・メッセージ
- メモリ・オーバーフロー・メッセージ
- リンカ内部メモリ

XLINK 警告メッセージ

XLINK 警告メッセージは、リンカが誤ったものを検出すると表示されます。しかし、生成されるコードは正しいこともあります。

XLINK エラー・メッセージ

XLINK エラー・メッセージは、リンカが誤ったものを検出すると生成されます。リンク処理は、Always generate output (-B) オプションが指定されていない場合には、中止されます。生成されるコードにはほとんどの場合、欠陥が含まれています。

致命的な XLINK エラー

致命的な XLINK エラー・メッセージは、リンク処理を中止します。致命的な XLINK エラーは、リンク処理を継続しても無意味な、すなわち障害から復帰できないような場合に発生します。

メモリ・オーバーフロー・メッセージ

XLINK は、メモリ・ベースのリンカです。メイン・メモリ容量の小さなシステムで実行されたり、あるいは非常に大きなソース・ファイルが使用される場合、XLINK はメモリからオーバーフローすることがあります。これは、次のメッセージによって確認できます。

```
* * * LINKER OUT OF MEMORY * * *
Dynamic memory used: nnnnnn bytes
```

このエラーが発生した場合、その解決方法はシステム・メモリを追加するか、ファイル・バウンド処理を可能にするかのどちらかです。-t オプションを使用して、メモリの内容を保存することもできます。

XLINK 内部エラー

リンキング中には、何回も内部一貫性チェックが実行されます。これらのチェックが1つでも不合格になると、リンカは不良内容の簡単な説明を発行した後に終了します。通常、このようなエラーは発生しませんが、万一発生した場合は、弊社技術サポート・グループに連絡してください。その場合、問題に関する全ての情報をお知らせください。できれば、内部エラーを発生したプログラムのコピー・ディスクも提供してください。

エラー・メッセージ

オブジェクト・ファイルの破損を示したメッセージが表示された場合、中断されたアセンブリやコンパイルは無効なオブジェクト・ファイルを生成することがあります。よって、不良ファイルは再度アセンブルあるいはコンパイルしてください。

No.	エラー・メッセージ	処置
0	Format chosen cannot support banking	この出力フォーマットはバンクをサポートできません。
1	Corrupt file. Unexpected end of file in module *module* (file) encountered	リンカは直ちに異常終了します。再度コンパイル／アセンブルするか、リンカとCコンパイラとの互換性をチェックしてください。
2	Too many errors encountered (>100)	リンカは直ちに異常終了します。

No.	エラー･メッセージ	処置
3	**Corrupt file. Checksum failed in module** *module (file)*. **Linker checksum is** *linkcheck*, **module checksum is** *modcheck*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
4	**Corrupt file. Zero length identifier encountered in module** *module (file)*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
5	**Address type for CPU incorrect. Error encountered in module** *module (file)*	リンカは直ちに異常終了します。 ファイルとライブラリをチェックして ください。
6	**Program module** *module* **redeclared in file** *file* **Ignoring second module**	-B オプション（ダンプ出力を行う）が指定 されていないと、XLINK はコードを作成 しません。
7	**Corrupt file. Unexpected UBROF - format end of file encountered in module** *module (file)*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
8	**Corrupt file. Unknown or misplaced tag encountered in module** *module (file)*. **Tag** *tag*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
9	**Corrupt file. Module** *module* **start unexpected in file** *file*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
10	**Corrupt file. Segment no.** *segno* **declared twice in module** *module (file)*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
11	**Corrupt file. External no.** *ext no* **declared twice in module** *module (file)*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。

No.	エラー・メッセージ	処置
12	**Unable to open file** *file*	リンカは直ちに異常終了します。環境変数 XLINK_DFLTDIR を調べてください。
13	**Corrupt file. Error tag encountered in module** *module (file)*	UBROF エラータグが見つかりました。リンカは直ちに異常終了します。再度コンパイル／アセブルしてください。
14	**Corrupt file. Local** *local* **defined twice in module** *module (file)*	リンカは直ちに異常終了します。再度コンパイル／アセンブルしてください。
15		（このエラーメッセージは削除されました。）
16	**Segment** *segment* **is too long for segment definition**	定義されているセグメントがこのセグメント用に予約されているメモリ領域に入りません。リンカは直ちに異常終了します。
17	**Segment** *segment* **is defined twice in segment definition -Zsegdef**	リンカは直ちに異常終了します。
18	**Range error in module** *module (file)*, **segment** *segment* **at address** *address*. **Value** *value*, **in tag** *tag*, **is out of bounds**	アドレスが CPU のアドレス範囲を超えています。このエラー・メッセージで与えられている情報を用いて問題の原因を特定してください。このチェックは、-R オプションにより抑制されます。
19	**Corrupt file. Undefined segment referenced in module** *module (file)*	リンカは直ちに異常終了します。再度コンパイル／アセンブルしてください。

No.	エラー・メッセージ	処置
20	Undefined external referenced in module *module (file)*	リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
21	Segment *segment* in module *module* does not fit bank	セグメントが長過ぎます。リンカは直ちに異常終了します。
22	Paragraph no. is not applicable for the wanted CPU. Tag encountered in module *module (file)*	リンカは直ちに異常終了します。 .xcl ファイル中のパラグラフ番号を消去してください。
23	Corrupt file. T_REL_FI_8 or T_EXT_FI_8 is corrupt in module *module (file)*	タグ T_REL_F1_8 または T_EXT_F1_8 に問題があります。リンカは直ちに異常終了します。 再度コンパイル／アセンブルしてください。
24	Segment *segment* overlaps segment *segment*	セグメント同士がオーバラップして、つまり同じアドレス上にあるコードをもっています
25	Corrupt file. Unable to find module *module (file)*	モジュールが見つかりません。リンカは直ちに異常終了します。
26	Segment *segment* is too long	プログラムが飛び抜けて大きい場合以外、このエラーは発生しません。リンカは直ちに異常終了します。
27	Entry ***entry*** in module *module (file)* redefined in module *module (file)*	同じ名前のエントリが複数あります。リンカは直ちに異常終了します。
28	File *file* is too long	プログラムが飛び抜けて大きい場合以外、このエラーは発生しません。リンカは直ちに異常終了します。
29	No object file specified in command-line	リンクするものが何もありません。リンカは直ちに異常終了します。
30	Option *-option* also requires the *-option* ***option***	リンカは直ちに異常終了します。

No.	エラー・メッセージ	処置
31	Option *-option* cannot be combined with the *-option* option	リンカは直ちに異常終了します。
32	Option *-option* cannot be combined with the *-option* option and the *-option* option	リンカは直ちに異常終了します。
33	Faulty value *val*, (range is 10-150)	ページの設定に問題があります。リンカは直ちに異常終了します。
34	Filename too long	ファイル名が 255 キャラクタを超えています。リンカは直ちに異常終了します。
35	Unknown flag *flag* in cross reference option *option*	リンカは直ちに異常終了します。
36	Option *op* does not exist	リンカは直ちに異常終了します。
37	- not succeeded by character	オプションの最初を示す -の後にキャラクタがありません。リンカは直ちに異常終了します。
38	Option *option* multiply defined	リンカは直ちに異常終了します。
39	Illegal character specified in option *op*	リンカは直ちに異常終了します。
40	Argument expected after option *op*	このオプションには引数が必要です。リンカは直ちに異常終了します。
41	Unexpected '-' in option *op*	リンカは直ちに異常終了します。
42	Faulty *symbol* definition - D*symbol definition*	文法が不当です。リンカは直ちに異常終了します。
43	Symbol in symbol definition too long	シンボル名が 255 キャラクタを超えています。リンカは直ちに異常終了します。

No.	エラー・メッセージ	処置
44	**Faulty value** *val*, **(range 80-300)**	カラムの設定に問題があります。 リンカは直ちに異常終了します。
45	**Unknown CPU** *CPU* **encountrered in** *context*	リンカは直ちに異常終了します。
46	**Undefined external** *external* **referred** *in module (file)*	外部参照用のエントリが見つかりません。
47	**Unknown format** *format* **encountered in** *context*	リンカは直ちに異常終了します。
48		(このエラーメッセージは削除されました。)
49		(このエラーメッセージは削除されました。)
50	**Paragraph no. not allowed for this CPU, encountered in option** *option*	リンカは直ちに異常終了します。 宣言中でパラグラフ番号は使用できません。
51	*Input base* **or decimal value expected in option** *option*	リンカは直ちに異常終了します。
52	**Overflow on value in option** *option*	リンカは直ちに異常終了します。
53	**Parameter exceeded 255 characters in extended command line file** *file*	リンカは直ちに異常終了します。
54	**Extended command line file** *file* **is empty**	リンカは直ちに異常終了します。

No.	エラー・メッセージ	処置
55	Extended command line variable XLINK_ENVPAR is empty	リンカは直ちに異常終了します。
56	Overlapping ranges in segment definition *segment def*	リンカは直ちに異常終了します。
57	No CPU defined	コマンドラインあるいは XLINK_CPU に CPU が定義されていません。リンカは直ちに異常終了します。
58	No format defined	コマンドラインあるいは XLINK_FORMAT にフォーマットが定義されていません。リンカは直ちに異常終了します。
59	Revision no. for file is incompatible with XLINK revision no.	リンカは直ちに異常終了します。このエラーが再コンパイル／アセンブリ後に発生した場合は、XLINK のバージョンが正しくない可能性があります。マスタ ディスクを調べてください。
60	Segment *segment* defined in bank definition and segment definition	リンカは直ちに異常終了します。
61		(このエラーメッセージは削除されました。)
62	File *file* multiply defined in command line	リンカは直ちに異常終了します。
63	Trying to pop an empty stack in module *module (file)*	リンカは直ちに異常終了します。再度コンパイル／アセンブルしてください。

No.	エラー・メッセージ	処置
64	**Module** *module (file)* **has not the same debug type as the other modules**	リンカは直ちに異常終了します。
65	**Faulty replacement definition –e** *replacement* **definition**	文法が不当です。リンカは直ちに異常終了します。
66	**Function with F-index** *index* **has not been defined before indirect reference in module** *module (file)*	定義されていないモジュールを間接的にコールしています。関数の宣言が抜けている可能性があります。
67	**Function** *name* **has same F-index as** *function-name*, **defined in module** *module (file)*	ファイルが壊れている可能性があります。ファイルを再度コンパイルしてください。
68	**External function** *name* **in module** *module (file)* **has no global definition**	他にエラーが見つからなければ、アセンブラをコールしている C プログラムで$DEFINE アセンブリ言語サポート指定子を用いた宣言が抜けているのが原因です。これはリンカにそが抜けていの関数に必要なメモリを知らせるために必要な宣言です。
69	**Indirect or recursive function** *name* **in module** *module (file)* **has parameters or auto variables in nondefault memory**	再帰的あるいは間接的にコールされた関数名が拡張された言語メモリ識別子(bit,data,idata など)を使用して使用が許されていない非デフォルトのメモリを指示しています。 間接的にコールされる関数へのパラメータは使用しているメモリ モデルのデフォルトのメモリ領域になければなりません。また、再帰関数の場合は、ローカル変数とパラメータの両方がデフォルトのメモリになければなりません。

No.	エラー・メッセージ	処置
70		(このエラーメッセージは削除されました。)
71	**Segment** *name* **is incorrectly defined (in a bank definition, has wrong segment type or mixed segment types)**	このエラーは前に定義されたセグメントの使用法を間違った場合に発生します。また、このエラーは既に定義されたリンカ制御ファイルを変更しようとしたときにも発生します。
72	**Segment** *name* **must be defined in a -Z definition**	リンカ (普通は C のシステム制御に必要なセグメント) ファイルにセグメントが抜けているか、スペル ミス (セグメント名では大文字か小文 字かで意味が変わります) かのいずれかが原因です。
73	**Label ?ARG_MOVE not found (recursive function need it)**	ライブラリにはこのラベルを含むモジュールが必要です。このラベルを削除した場合は、リストアしてください。
74	**There was as error when writing to file** *file*	リンカまたはお使いのホスト システムが壊れています。あるいは両方に問題があります。
75	**SFR address in module** *module (file)***, segment** *segment* **at address** *address***, value** *value* **is out of bounds**	SFR が誤ったアドレスに対して定義されています。定義内容を変更してください。
76	**Absolute segments overlap module** *module*	リンカが *module* 内に互いにオーバラップする複数のアブソリュートセグメントを見つけました。
77	**Absolute segment in module** *module (file)* **overlaps absolute segment in module** *module (file)*	*module*（ファイル）と *module*（ファイル）内に互いにオーバラップする複数の絶対セグメントが見つかりました。

No.	エラー・メッセージ	処置
78	**Absolute segment in module** *module (file)* **overlaps segment** *segment*	*module*（ファイル）内に再配置可能なセグメントにオーバラップする絶対セグメントが見つかりました。
79	**Fautly allocation definition -a***definition*	オーバレイ制御定義の中にエラーが見つかりました。
80	**Symbol in allocation definition (-a) too long**	-a コマンドのシンボルが長過ぎます。
81	**Unknown flag in extended format** *option*	フラグを調べてください。
82	**Conflict in segment** *'name'* **Mixing overlayable and not overlayable segment parts.**	8051 および変換された PL/M モードでのみこのエラーが発生します。
83	**The overlayable segment** *'name'* **may not be banked**	このエラーは 8051 および変換された PL/M モードでのみ発生します。
84	**The overlayable segment** *'name'* **must be of relative type.**	このエラーは 8051 および変換された PL/M モードでのみ発生します。
85	**The far/farc segment** *name* **in module** *module (file)***, is larger than** *size*	セグメント*name*がfarセグメントには大きすぎます。
86		（このエラーメッセージは削除されました。）
87	**Function with F-index** *i* **has not been defined before tiny_func referenced in module** *module (file)*	すべてのtiny関数がモジュール内において使用される前に定義されているか、調べてください。

No.	エラー・メッセージ	処置
88	Wrong library used (compiler version or memory model mismatch). Problem found in *module (file).* Correct library tag is *tag*	このコンパイラからのコードは、合致しているライブラリを必要としています。以前または以後のバージョンのコンパイラに属すライブラリが使用されている可能性があります。
92	Cannot use this format with this cpu	いくつかのフォーマットがCPU特有の情報を必要としていて、あるCPUのみをサポートしています。
93	Non-existent warning number *no*, (valid numbers are 0-*max*)	存在しない警告を抑制しようとすると、このエラーが発生します。
94	Unknown flag *x* in local symbols option -n*x*	文字*x*がローカル・シンボル・オプション中の有効なフラッグではありません。
95	Module *mod (file)* uses source file references, which are not available in UBROF 5 output numbers are 0-max)	このUBROF 5の出力を生成するときには、この特性はリンカを通ることができません。
96	Unmatched -! comment in extended command file	.xclファイル中に奇数個の-!(コメント)オプションがあります。
97	Unmatched -! comment in extended command line variable XLINK_ENVPAR	上記と同様ですが、環境変数XLINK_ENVPARについてです。
98	Unmatched /* comment in extended command file numbers are 0-max)	.xclファイル中に適合する */ が見つかりません。
99	Syntax error in segment definition *option*	オプション中に構文エラーがあります。

No.	エラー・メッセージ	処置
100	Segment name too long: "*seg*" in *option*	セグメント名が最大長 (255文字) を越えました。
101	Segment already defined: "*seg*" in option	このセグメントがすでにセグメント定義オプションで記述されています。
102	No such segment type: *option*	与えられたセグメント型が不正です。
103	Range must be closed in *option*	-Pオプションは、すべてのメモリ範囲に終端があることを要求します。
104	Failed to fit all segments into specified ranges. Problem discovered in segment *seg*	リンカで使用されているパッキングのアルゴリズムは、すべてのセグメントに合うように管理していません。
105	**Recursion not allowed for this system. Check module map for recursive functions**	使用されているランタイム・モデルは、再帰性をサポートしていません。リンカで再帰性があると定められたそれぞれの関数は、リンカ・リストファイルのモジュール・マップ部分にそのようにマークされています。
106	Syntax error or bad argument in *option*	与えられたコマンドライン引数を解析する際にエラーがありました。
107	Banked segments do not fit into the number of banks specified	リンカは、バンクセグメントのすべてが与えられたバンクにフィットするようには、管理していません。
108	Cannot find function *function* mentioned in –a#	間接呼び出しオプションで指定されたすべての関数は、リンクされるプログラム中に存在していなければなりません。
109	Function *function* mentioned as callee in –a# is not indirectly called	実際に間接的に呼び出される関数のみが、間接呼び出しオプションでそのように指定できます。
110	Function *function* mentioned as caller in –a# does not make indirect calls	実際に間接的呼び出しをする関数のみが、間接呼び出しオプションでそのように指定できます。
111	The file *file* is not a UBROF	ファイルの内容が、XLINKの読み込み可能な形式ではありません。

No	エラー・メッセージ	処置
112	The module *module* is for an unknown cpu (tid = *tid*). Either the file is corrupt or you need a later version of XLINK	使用されているXLINKのバージョンが、コンパイルまたはアセンブルされたファイルのCPUについての情報を持ちません。
113	Corrupt input file: "*symptom*" in *module (file)*	示されているファイルが破損している可能性があります。このメッセージは、何かの理由でファイルが生成後に壊れているか、または使用されたコンパイラ／アセンブラに問題があったを示しています。後者の場合には、弊社にご連絡ください。
114		(このメッセージは存在しません。)
115	Unmatched '"' in extended command file or XLINK_ENVAR	拡張コマンドファイル、または環境変数XLINK_ENVPARを解析する際に、XLINKは一致しない引用符を検知しました。引用符を伴ったファイル名に対しては、その引用符の前にバックスラッシュ(\)を置く必要があります。たとえば、 c:\iar\"A file called \"file\"" は、XLINKからは、c:\iar\directory にある A file called "file" というファイルで探索されます。
116	Definition of *symbol* in module *module1* is not compatible with definition of *symbol* in module *module2*	シンボル*symbol*が、少なくとも1つのモジュールで試験的に定義されています。試験的な定義は、他の定義に合致している必要があります。
117	Incompatible runtime modules. Module *module1* specifies that *attribute* must be *value1*, but module *module2* has the value *value2*	これらのモジュールは、互いにリンクすることはできません。これらは互換性のない、ランタイム・モジュールになるような設定でコンパイルされました。
118	Incompatible runtime modules. Module *module1* specifies that *attribute* must be *value*, but module *module2* specifies no value for this attribute.	

警告メッセージ

下表に、リンカの警告メッセージをリストします。

No.	警告メッセージ	処置
0	Too many warnings	警告の数が制限を超えています。
1	Error tag encountered in module *module (file)*	ファイル *file* ロード時に UBROF エラータグが見つかりました。これは、ファイルが壊れていることを意味します。
2	Symbol *symbol* is redefined in command-line	シンボルが再定義されました。
3	Type conflict. Segment *segment*, in module *module*, is incompatible with earlier segment(s) of the same name	名前が同じセグメントは同じタイプでなければなりません。
4	Close/open conflict. Segment *segment*, in module *module*, is incompatible with earlier segment of the same name	同じ名前のセグメントはすべてオープンするかクローズするかどちらかにしなければなりません。
5	Segment *segment* cannot be combined with previous segment	複数のセグメントを組み合わせることはできません。
6	Type conflict for external/entry *entry*, in module *module*, against external/entry in module *module*	エントリと対応する外部宣言は同じタイプにしなければなりません。
7	Module *module* declared twice, once as program and once as library. Redeclared in file *file*, ignoring library module	同じモジュールが2つ定義されています。プログラムモジュールはリンクされ、ライブラリの方は無視されます。
8		（この警告メッセージは削除されました。）

No.	警告メッセージ	処置
9	**Ignoring redeclared program entry in module** *module (file)*, **using entry from** *module1*	最初に見つけたプログラム エントリだけを選択します。
10	**No modules to link**	リンクするモジュールがありません。
11	**Module** *module* **declared twice as library. Redeclared in file** *file*, **ignoring second module**	最初のモジュールだけをリンクします。
12	**Using SFB in banked segment** *segment* **in module** *module* **(file)**	バンク セグメントでは SFB アセンブラ指定子が動作しないことがあります。
13	**Using SFE in banked segment** *segment* **in module** *module* **(file)**	バンク セグメントでは SFE アセンブラ指定子が動作しないことがあります。
14	**Entry** *entry* **duplicated. Module** *module (file)* **loaded, module** *module (file)* **discarded**	条件つきでロードされたモジュール、つまりライブラリ モジュールまたは条件つきでロードされたプログラム モジュール (-C オプション) の中に重複するエントリがあります。
15	**Predefined type sizing mismatch between modules** *module (file)* **and** *module (file)*	このモジュールは、基本的な C の異なるサイズ (例: integer, double) のように、既に定義されたタイプに対して別のオプションでコンパイルしています。
16	**Function** *name* **in module** *module (file)* **is called from two function trees (with roots** *name1* **and** *name2***)**	さらにフォアグランド プログラムから実行される別の関数をコールしている割り込み関数があります。
17	**Segment** ***name*** **is too large or placed at wrong address**	指定したセグメントが指定したメモリ領域内の利用可能なアドレス スペースをオーバランしています。オーバランしている箇所を見つけるには、ダミー リンクを行い指定したセグメントの開始アドレスを最低位アドレスに移して、リンカ マップ ファイルを調べてください。アドレスを修正し終わったら、再度リンクしてください。
18	**Segment** *segment* **overlaps segment** *segment*	2つのセグメントがオーバラップしています。セグメント配置オプションの指定をチェックしてください。

No.	警告メッセージ	処置
19	**Absolute segments overlaps in module** *module (file)*	**モジュール** *module* **に互いにオーバラップする複数のセグメントが見つかりました。**
20	**Absolute segment in module** *module (file)* **overlaps absolute segment in module** *module (file)*	*module*（ファイル）**と** *module*（ファイル）**内に互いにオーバラップする複数の絶対セグメントが見つかりました。ORG 指定子を変更してください。**
21	**Absolute segment in module** *module (file)* **overlaps segment** *segment*	*module*（ファイル）**内に再配置可能なセグメントにオーバラップする絶対セグメントが見つかりました。ORG 指定子か-Z 再配置コマンドのどちらかを変更してください。**
22	**Interrupt function** ***name*** **in module** *module (file)* **is called from other functions**	**割り込み関数がコールされた可能性があります。**
23	*limitation specific warning*	**選択された出力形式または利用可能な情報におけるいくつかの制限のために、**XLINK**は正しい出力を行うことができません。それぞれに特有の制限に対して1つの警告のみが与えられます。**
24	*num* **counts of** *warning* **total**	**実際に発生した**23**のタイプのそれぞれの警告に対して、概要が最後に与えられます。**
25	**Using ¥"-Y#¥" discards and distorts debug information. Use with care. If possible, find an updated debugger that can read modern UBROF**	UBROF**フォーマットの修飾子 -Y# を用いることは、薦められません。**
26	**No reset vector found**	**リセット･ベクトルが見つからないため、**78400**プロセッサに対する**XCOFF**出力形式の**LOCATION**の設定を決定できませんでした。**
27	**No code at start address found in reset vector**	**リセット･ベクトルで指定されているアドレスにコードが見つからなかったため、**78400**プロセッサに対する**XCOFF**出力形式の**LOCATION**の設定を決定できませんでした。**

No.	警告メッセージ	処置
28	Parts of segment *name* are initialized, parts not	リンク結果がPROM化可能ならば、この警告は無用です。
29	Parts of segment *name* are initialized, even though it is of type *type* (and thus not promable)	リンク結果がPROM化可能ならば、DATAメモリの初期化は無用です。
30	Module *name* is compiled for *cpu1*, expected cpu2 name *cpu1*.	CPU *cpu2*用の実行ファイルを作成しようとしましたが、モジュール*name*はCPU *cpu1*用にコンパイルされています。
31	Modules have been compiled with possibly incompatible settings: *more info*	モジュールの内容では、これらは互換性がありません。
32	Format option set more than once. Using format *format*	フォーマット・オプションは、1回しか使用できません。リンカは*format*フォーマットを使用します。
33	Using -r overrides format option. Using UBROF	-rオプションはUBROFフォーマットとC-SPYライブラリ・モジュールを指定します。これは、いかなる-F〔フォーマット〕オプションをも上書きします。
34	The 20 bit segmented variant of the INTEL EXTENDED format cannot represent the addresses specified. Consider using -Y1 (32 bit linear addressing)	プログラムが0xFFFFFより高いアドレスを使用していて、選択された形式のセグメントのバリアントがこれをあつかうことができません。リニア・アドレス・バリアントは、全32ビット・アドレスをあつかえます。
35	There is more than one definition for the struct/union type with the tag *tag*	同じtagをもった、2つ以上の異なった構造体／共用体が、プログラム中に存在します。もし、意図的でないならば、宣言が多少異なっているものと思われます。また、型の衝突についての警告〔warning 6〕が2つ以上ある可能性がかなりあります。もしこれが意図的なら、この警告をなくすことを考慮してください。

No	警告メッセージ	処置
36	There are indirectly called functions doing indirect calls. This can make the static overlay system unreliable	XLINKは、この場合、どの関数がどの関数を呼び出すかを知りません。このことは、静的オーバーレイが安全かどうかが確認できないことを意味しています。
37	More than on interrupt function makes indirect calls. This can make the static overlay system unreliable. Using –ai will avoid this	関数がすでに実行中の割り込み関数から呼び出された場合、そのパラメータと引数は上書きされます。
38	There are indirect calls both from interrupt and from the main program. This can make the static overlay system unreliable. Using –ai will avoid this	関数がすでに実行中の割り込み関数から呼び出された場合、そのパラメータと引数は上書きされます。
39	The function *function* in module *module (file)* does not appear to be called. No static overlay area will be allocated for its params and locals	XLINKに関する限り、その関数に対する呼び出しがなく、したがってそのパラメータとローカル変数用のスペースは不要です。XLINKにどのような場合でもスペースを割り当てるには、-a (関数)を使用します。
40	The module *module* contains obsolete type information that will not be checked by the linker	この種類の型情報は、1988年に置き換えられました。
41	The function *function* in module *module (file)* makes indirect calls but is not mentioned in the left part of any –a# declaration	-a# 間接呼び出しオプションがいくつか存在する場合、これらのオプションはともに完全な状況を指定します。
42		(この警告メッセージは存在しません。)

No	警告メッセージ	処置
43	The function *function* in module *module (file)* is indirectly called but is not mentioned in the right part of any –a# declaration	-a# 間接呼び出しオプションがいくつか存在する場合、これらのオプションはともに完全な状況を指定します。
44	C library routine localtime failed. Timestamps will be wrong	XLINKは正しい時間を決定することができません。このことは第一にリストファイルの日付に影響を与えます。この問題は、日付が2038年以降の場合に、あるホストのプラットフォームで見出されています。
45	Memory attribute info mismatch between modules *module1 (file1)* and *module2 (file2)*	与えられたモジュールにおけるUBROF 7メモリ属性情報が同じではありません。
46	External function function in module (file) has no global definition	この警告はエラー68を置き換えます。
47	Range error in *module (file)*, segment *segment* at address *address*. Value *value*, in tag *tag*, is out of bounds *bounds*	この警告は、-Rwが指定されたときに、エラー18を置き換えます。

第１７章 XLIB 診断

この章では、XLIB ライブラリアンが発生するメッセージの一覧を示します。

XLIB メッセージ

以下の表は、XLIB のメッセージの一覧を提供しています。エラー表示されたコマンドがオブジェクトファイルを変更することはありません。

No.	エラー・メッセージ	処置
1	Bad object file, EOF encountered	オブジェクト ファイルが不正、または空です。アセンブリ／コンパイルが異常終了した可能性があります。
2	Unexpected EOF in batch file	コマンド ファイルの最後のコマンドは EXIT でなければなりません。
3	Unable to open file *file*	コマンド ファイルをオープンできません。また、ON-ERROR-EXIT が指定されている場合は、ファイルのオープンに関する何らかの処理に失敗するとこのメッセージが表示されます。
4	Variable length record out of bounds	オブジェクト モジュールが不正です。アセンブリが異常終了した可能性があります。
5	Missing or non-default parameter	直接モードでパラメータが抜けています。
6	No such CPU	このエラーが発生すると、選択できる CPU のリストが表示されます。
7	CPU undefined	DEFINE-CPU はオブジェクト ファイルの処理を開始するする前に指定する必要があります。このエラーが発生すると、CPU のリストが表示されます。
8	Ambiguous CPU type	このエラーが発生すると、CPU のリストが表示されます。
9	No such command	HELP コマンドをご使用ください。

No.	エラー・メッセージ	処置
10	Ambiguous command	HELP コマンドをご使用ください。
11	Invalid parameter(s)	パラメータの数が多すぎるか、パラメータのスペルに誤りがあります。
12	Module out of sequence	オブジェクト モジュールが不正です。アセンブリが異常終了した可能性があります。
13	Incompatible object, consult ditributor!	オブジェクト ファイルが不正で、アセンブリが異常終了した可能性があります。あるいは、お使いのアセンブラ／コンパイラのバージョンとXLIB のバージョンとの間に互換性がありません。
14	Unknown tag: hh	オブジェクト モジュールが不正です。アセンブリが異常終了した可能性があります。
15	Too many errors	エラーの数が 32 を超えると XLIB は異常終了します。
16	Assembly/compilation error?	T_ERROR タグが見つかりました。プログラムを編集し再度アセンブル／コンパイルしてください。
17	Bad CRC, hhhh expected	オブジェクト モジュールが不正です。アセンブリが異常終了した可能性があります。
18	Can't find module: xxxxx	LIST-MOD ファイルで使用できるモジュールを調べてください。
19	Module expression out of range	モジュール式が 1 より小さいか、$より大きくなっています。
20	Bad syntax in module expression: xxxxx	文法が不当です。
21	Illegal insert sequence	INSERT-MODULES コマンドで指定した dest は start-end 範囲外になければなりません。
22	<End module> found before <Start module>!	ソース モジュールの範囲は下位、上位の順で指定しなければなりません。
23	Before or after!	INSERT-MODULES コマンドの BEFORE|AFTER 識別子が不当です。

No.	エラー・メッセージ	処置
24	Corrupt file, error occurred in *tag*	オブジェクト ファイル *tag* の中にエラーが見つかりました。再度アセンブリ／コンパイルしてみてください。それでもエラーが発生するようでしたら、弊社の技術サポートまでご連絡ください。
25	*File* is write protected	ファイル *file* が書き込み禁止で書き込めません。
26	Non-matching replacement module ***name*** found in source file	ソース ファイルの中に目的ファイルの内に対応するエントリをもたないモジュール *name* が見つかりました。

IMA アセンブラ・プログラミング・ガイド
（740 ファミリ） 第 2 版

RENESAS
ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668